NAME

pthread_sigmask, thr_sigsetmask – change and/or examine calling thread's signal mask

SYNOPSIS

POSIX

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

#include <pthread.h>
#include <signal.h>
int pthread_sigmask(int *how***, const sigset_t** **set,* **sigset_t** **oset***);**

Solaris

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

#include <thread.h>
#include <signal.h>
int thr_sigsetmask(int *how***, const sigset_t** **set,* **sigset_t** **oset***);**

MT-LEVEL

MT-Safe

Async-Signal-Safe

DESCRIPTION

pthread_sigmask( ) and **thr_sigsetmask( )** changes and/or examines a calling thread's signal mask. Each thread has its own signal mask. A new thread inherits the calling thread's signal mask and priority, however, pending signals are not inherited. Signals pending for a new thread will be empty.

If the value of the argument *set* is not **NULL,** *set* points to a set of signals that can modify the currently blocked set. If the value of *set* is **NULL**, the value of *how* is insignificant and the thread's signal mask is unmodified; thus, **pthread_sigmask( )** or **thr_sigsetmask( )** can be used to inquire about the currently blocked signals.

The value of the argument *how* specifies the method in which the set is changed. *how* takes one of the following values:

SIG_BLOCK *set* corresponds to a set of signals to block. They are added to the current signal mask.

SIG_UNBLOCK *set* corresponds to a set of signals to unblock. These signals are deleted from the current signal mask.

SIG_SETMASK *set* corresponds to the new signal mask. The current signal mask is replaced by *set*.

If the value of *oset* is not **NULL**, it points to the location where the previous signal mask is stored.

RETURN VALUES

Zero is returned upon successful completion; otherwise, a non-zero value indicates an error.

ERRORS

If any of the following conditions occur, **pthread_sigmask( )** or **thr_sigsetmask( )** fails and returns the corresponding value:

EINVAL *set* is not **NULL** and the value of *how* is not defined.

If any of the following conditions are detected, **pthread_sigmask( )** or **thr_sigsetmask( )** fails and returns the corresponding value:

EFAULT *set* or *oset* are not valid addresses.

EXAMPLES

The following example shows how to create a default thread that can serve as a signal catcher/handler with its own signal mask. *new* will have a different value than the creator's signal mask.

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT    /* basic first 3-lines for threads */
#include <pthread.h>
#include <thread.h>

thread_t user_threadID;
sigset_t new;
void *handler(), interrupt();

main( int argc, char *argv[] ) {
 test_argv(argv[1]);

 sigemptyset(&new);
 sigaddset(&new, SIGINT);
 switch(*argv[1]) {

  case '0':   /* POSIX */
   pthread_sigmask(SIG_BLOCK, &new, NULL);
   pthread_create(&user_threadID, NULL, handler, argv[1]);
   pthread_join(user_threadID, NULL);
   break;

  case '1':   /* Solaris */
   thr_sigsetmask(SIG_BLOCK, &new, NULL);
   thr_create(NULL, 0, handler, argv[1], 0, &user_threadID);
   thr_join(user_threadID, NULL, NULL);
   break;
  }  /* switch */

  printf("thread handler, # %d, has exited\n",user_threadID);
  sleep(2);
  printf("main thread, # %d is done\n", thr_self());
} /* end main */

struct sigaction act;

void *
handler(char argv1[])
```

```
{
 act.sa_handler = interrupt;
 sigaction(SIGINT, &act, NULL);
 switch(*argv1) {
  case '0':    /* POSIX */
   pthread_sigmask(SIG_UNBLOCK, &new, NULL);
   break;
  case '1':  /* Solaris */
   thr_sigsetmask(SIG_UNBLOCK, &new, NULL);
   break;
 }
 printf("\n Press cntrl-C to deliver SIGINT signal to the process\n");
 sleep(8);  /* give user time to hit cntrl-C */
}

void
interrupt(int sig)
{
 printf("thread %d caught signal %d\n", thr_self(), sig);
}

void test_argv(char argv1[])    {
 if(argv1 == NULL)  {
   printf("use 0 as arg1 to use thr_create();\n \
   or use 1 as arg1 to use pthread_create()\n");
   exit(NULL);
 }
}
```

Since POSIX threads and Solaris threads are fully compatible even within the same process, this example uses **pthread_create**(3T) if you execute **a.out 0**, or **thr_create**(3T) if you execute **a.out 1**.

Here's an explanation of the above example:

- **sigemptyset**(3C) initializes a null signal set, *new*. **sigaddset**(3C) packs the signal, **SIGINT,** into that new set.
- Either **pthread_sigmask( )** or **thr_sigsetmask( )** is used to mask the signal, **SIGINT,** (cntrl-C), from the calling thread, which is **main( )**. The signal is masked to guarantee that only the new thread will receive this signal.
- **pthread_create( )** or **thr_create( )** creates the signal-handling thread.
- Using **pthread_join**(3T) or **thr_join**(3T), **main( )** then waits for the termination of that signal-handling thread, whose ID number is *user_threadID*; after which, **main( )** will **sleep**(3C) for 2 seconds, and then the program terminates.

- The signal-handling thread, *handler*:
  - Assigns the handler *interrupt( )* to handle the signal SIGINT , via the call to **sigaction**(2).
  - Resets its own signal set to NOT BLOCK the signal, **SIGINT.**
  - Sleeps for 8 seconds to allow time for the user to deliver the signal, **SIGINT,** by pressing the cntrl-C keys.

In the example, the *handler* thread served as a signal-handler while also taking care of activity of its own (in this case, sleeping, although it could have been some other activity). A thread could be completely dedicated to signal-handling, simply by waiting for the delivery of a selected signal by blocking with **sigwait**(2). Thus, the two subroutines in the previous example, *handler( )* and *interrupt( )*, could have been replaced with the following routine:

```
void *
handler()
{ int signal;
  printf("thread %d is waiting for you to press the cntrl-C keys\n", thr_self());
  sigwait(&new, &signal);
  printf("thread %d has received the signal %d \n", thr_self(), signal);
}
 /* pthread_create( ) and thr_create( ) would use NULL instead of argv[1]
    for the arg passed to handler( ) */
```

In this routine, one thread is dedicated to catching and handling the signal specified by the set, *new*, which allows **main( )** and all of its other sub-threads, (created AFTER **pthread_sigmask( )** or **thr_sigsetmask( )** masked that signal), to continue uninterrupted. In fact, any use of **sigwait**(2) should be such that all threads block the signals passed to **sigwait**(2), at all times. Only the thread that calls **sigwait( )** will get the signals. Note that the call to **sigwait**(2) takes two arguments. See **sigwait**(2).

For this type of background dedicated signal-handling routine, you may wish to use a Solaris daemon thread by passing the argument, **THR_DAEMON,** to **thr_create**(3T).

SEE ALSO

sigaction(2), **sigprocmask**(2), **sigwait**(2), **sigsetops**(3C), **pthread_cancel**(3T), **pthread_create**(3T), **pthread_exit**(3T), **pthread_join**(3T), **pthread_kill**(3T), **pthread_self**(3T)

NOTES

It is not possible to block signals that cannot be ignored (see **sigaction**(2)). If using the threads library, it is not possible to block the signals **SIGLWP** or **SIGCANCEL,** which are reserved by the threads library. Additionally, it is impossible to unblock the signal **SIGWAITING,** which is always blocked on all threads. This restriction is quietly enforced by the threads library.

Using **sigwait**(2) in a dedicated thread allows asynchronous signals to be managed synchronously; however, **sigwait**(2) should never be used to manage synchronous signals. Synchronous signals (i.e., exceptions or traps) are sent by the process itself, such as **SIGFPE, pthread_kill**(3T), **pthread_exit**(3T), **pthread_cancel**(3T), **thr_kill**(3T), or

thr_exit(3T), rather than device interrupts or signals sent by other processes.

Synchronous signals are exceptions that are generated by a thread and are directed at the thread causing the exception. Since **sigwait( )** blocks waiting for signals, the blocking thread will not generate any synchronous signals.

If **sigprocmask**(2) is used in a multi-threaded program, it will be the same as if **thr_sigsetmask( )** or **pthread_sigmask( )** has been called. Note that POSIX leaves the semantics of the call to **sigprocmask**(2) unspecified in a multi-threaded process, so programs that care about POSIX portability should not depend on this semantic.

If a signal is delivered while a thread is waiting on a condition variable, the **cond_wait( )** will be interrupted and the handler will be executed. The handler should assume that the lock protecting the condition variable is held.

Although **pthread_sigmask( )** is Async-Signal-Safe with respect to the Solaris environment, this safeness is not guaranteed to be portable to other POSIX domains.