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

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'sigtimedwait.3p' command

$ man sigtimedwait.3p

SIGTIMEDWAIT(3P) POSIX Programmer's Manual SIGTIMEDWAIT(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

sigtimedwait, sigwaitinfo ? wait for queued signals

SYNOPSIS

```
#include <signal.h>
int sigtimedwait(const sigset_t *restrict set,
    siginfo_t *restrict info,
    const struct timespec *restrict timeout);
int sigwaitinfo(const sigset_t *restrict set,
    siginfo_t *restrict info);
```

DESCRIPTION

The sigtimedwait() function shall be equivalent to sigwaitinfo() except that if none of the signals specified by set are pending, sigtimed? wait() shall wait for the time interval specified in the timespec structure referenced by timeout. If the timespec structure pointed to by timeout is zero-valued and if none of the signals specified by set are pending, then sigtimedwait() shall return immediately with an er? ror. If timeout is the null pointer, the behavior is unspecified. If

the Monotonic Clock option is supported, the CLOCK_MONOTONIC clock shall be used to measure the time interval specified by the timeout ar? gument.

The sigwaitinfo() function selects the pending signal from the set specified by set. Should any of multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or be? tween multiple pending non-realtime signals, is unspecified. If no sig? nal in set is pending at the time of the call, the calling thread shall be suspended until one or more signals in set become pending or until it is interrupted by an unblocked, caught signal.

The sigwaitinfo() function shall be equivalent to the sigwait() func? tion, except that the return value and the error reporting method are different (see RETURN VALUE), and that if the info argument is non- NULL, the selected signal number shall be stored in the si_signo mem? ber, and the cause of the signal shall be stored in the si_code member. If any value is queued to the selected signal, the first such queued value shall be dequeued and, if the info argument is non-NULL, the value shall be stored in the si_value member of info. The system re? source used to queue the signal shall be released and returned to the system for other use. If no value is queued, the content of the si_value member is undefined. If no further signals are queued for the selected signal, the pending indication for that signal shall be reset.

RETURN VALUE

Upon successful completion (that is, one of the signals specified by set is pending or is generated) sigwaitinfo() and sigtimedwait() shall return the selected signal number. Otherwise, the function shall return a value of -1 and set errno to indicate the error.

ERRORS

The sigtimedwait() function shall fail if:

EAGAIN No signal specified by set was generated within the specified timeout period.

The sigtimedwait() and sigwaitinfo() functions may fail if:

EINTR The wait was interrupted by an unblocked, caught signal. It shall be documented in system documentation whether this error causes these functions to fail.

The sigtimedwait() function may also fail if:

EINVAL The timeout argument specified a tv_nsec value less than zero or greater than or equal to 1000 million.

An implementation should only check for this error if no signal is pending in set and it is necessary to wait.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

The sigtimedwait() function times out and returns an [EAGAIN] error. Application developers should note that this is inconsistent with other functions such as pthread_cond_timedwait() that return [ETIMEDOUT].

Note that in order to ensure that generated signals are queued and sig? nal values passed to sigqueue() are available in si_value, applications which use sigwaitinfo() or sigtimedwait() need to set the SA_SIGINFO flag for each signal in the set (see Section 2.4, Signal Concepts). This means setting each signal to be handled by a three-argument sig? nal-catching function, even if the handler will never be called. It is not possible (portably) to set a signal handler to SIG_DFL while set? ting the SA_SIGINFO flag, because assigning to the sa_handler member of struct sigaction instead of the sa_sigaction member would result in un? defined behavior, and SIG_DFL need not be assignment-compatible with sa_sigaction. Even if an assignment of SIG_DFL to sa_sigaction is ac? cepted by the compiler, the implementation need not treat this value as special?it could just be taken as the address of a signal-catching function.

RATIONALE

Existing programming practice on realtime systems uses the ability to pause waiting for a selected set of events and handle the first event that occurs in-line instead of in a signal-handling function. This al?

lows applications to be written in an event-directed style similar to a state machine. This style of programming is useful for largescale transaction processing in which the overall throughput of an applica? tion and the ability to clearly track states are more important than the ability to minimize the response time of individual event handling.

It is possible to construct a signal-waiting macro function out of the realtime signal function mechanism defined in this volume of POSIX.1?2017. However, such a macro has to include the definition of a generalized handler for all signals to be waited on. A significant por? tion of the overhead of handler processing can be avoided if the sig? nal-waiting function is provided by the kernel. This volume of POSIX.1?2017 therefore provides two signal-waiting functions?one that waits indefinitely and one with a timeout?as part of the overall real? time signal function specification.

The specification of a function with a timeout allows an application to be written that can be broken out of a wait after a set period of time if no event has occurred. It was argued that setting a timer event be? fore the wait and recognizing the timer event in the wait would also implement the same functionality, but at a lower performance level. Be? cause of the performance degradation associated with the user-level specification of a timer event and the subsequent cancellation of that timer event after the wait completes for a valid event, and the com? plexity associated with handling potential race conditions associated with the user-level method, the separate function has been included.

Note that the semantics of the sigwaitinfo() function are nearly iden? tical to that of the sigwait() function defined by this volume of POSIX.1?2017. The only difference is that sigwaitinfo() returns the queued signal value in the value argument. The return of the queued value is required so that applications can differentiate between multi? ple events queued to the same signal number.

The two distinct functions are being maintained because some implemen? tations may choose to implement the POSIX Threads Extension functions and not implement the queued signals extensions. Note, though, that

sigwaitinfo() does not return the queued value if the value argument is NULL, so the POSIX Threads Extension sigwait() function can be implemented as a macro on sigwaitinfo().

The sigtimedwait() function was separated from the sigwaitinfo() function to address concerns regarding the overloading of the timeout pointer to indicate indefinite wait (no timeout), timed wait, and immediate return, and concerns regarding consistency with other functions where the conditional and timed waits were separate functions from the pure blocking function. The semantics of sigtimedwait() are specified such that sigwaitinfo() could be implemented as a macro with a null pointer for timeout.

The sigwait functions provide a synchronous mechanism for threads to wait for asynchronously-generated signals. One important question was how many threads that are suspended in a call to a sigwait() function for a signal should return from the call when the signal is sent. Four choices were considered:

1. Return an error for multiple simultaneous calls to sigwait functions for the same signal.
2. One or more threads return.
3. All waiting threads return.
4. Exactly one thread returns.

Prohibiting multiple calls to sigwait() for the same signal was felt to be overly restrictive. The ``one or more'' behavior made implementation of conforming packages easy at the expense of forcing POSIX threads clients to protect against multiple simultaneous calls to sigwait() in application code in order to achieve predictable behavior. There was concern that the ``all waiting threads'' behavior would result in ``signal broadcast storms'', consuming excessive CPU resources by replicating the signals in the general case. Furthermore, no convincing examples could be presented that delivery to all was either simpler or more powerful than delivery to one.

Thus, the consensus was that exactly one thread that was suspended in a call to a sigwait function for a signal should return when that signal

occurs. This is not an onerous restriction as:

* A multi-way signal wait can be built from the single-way wait.
* Signals should only be handled by application-level code, as li? brary routines cannot guess what the application wants to do with signals generated for the entire process.
* Applications can thus arrange for a single thread to wait for any given signal and call any needed routines upon its arrival.

In an application that is using signals for interprocess communication, signal processing is typically done in one place. Alternatively, if the signal is being caught so that process cleanup can be done, the signal handler thread can call separate process cleanup routines for each por? tion of the application. Since the application main line started each portion of the application, it is at the right abstraction level to tell each portion of the application to clean up.

Certainly, there exist programming styles where it is logical to con? sider waiting for a single signal in multiple threads. A simple sig? wait_multiple() routine can be constructed to achieve this goal. A pos? sible implementation would be to have each sigwait_multiple() caller registered as having expressed interest in a set of signals. The caller then waits on a thread-specific condition variable. A single server thread calls a sigwait() function on the union of all registered signals. When the sigwait() function returns, the appropriate state is set and condition variables are broadcast. New sigwait_multiple() call? ers may cause the pending sigwait() call to be canceled and reissued in order to update the set of signals being waited for.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4, Signal Concepts, Section 2.8.1, Realtime Signals, pause(), pthread_sigmask(), sigaction(), sigpending(), sigsuspend(), sigwait()

The Base Definitions volume of POSIX.1?2017, <signal.h>, <time.h>

COPYRIGHT