NAME

sigaction – detailed signal management

SYNOPSIS

#include <signal.h>

int sigaction(int *sig*, **const struct sigaction** ∗*act*, **struct sigaction** ∗*oact***);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

sigaction( ) allows the calling process to examine or specify the action to be taken on delivery of a specific signal. (See **signal**(5) for an explanation of general signal concepts.)

sig specifies the signal and can be assigned any of the signals specified in **signal**(5) except **SIGKILL** and **SIGSTOP.** In a multi-threaded process, *sig* cannot be **SIGWAITING, SIGCANCEL,** or **SIGLWP.**

If the argument *act* is not **NULL,** it points to a structure specifying the new action to be taken when delivering *sig*. If the argument *oact* is not **NULL,** it points to a structure where the action previously associated with *sig* is to be stored on return from **sigaction( )**.

The **sigaction** structure includes the following members:

```
void        (*sa_handler)( );
void        (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t    sa_mask;
int         sa_flags;
```

sa_handler identifies the action to be associated with the specified signal, if the **SA_SIGINFO** flag (see below) is cleared in the **sa_flags** field of the sigaction structure. It may take any of the values specified in **signal**(5) or that of a user specified signal handler. If the **SA_SIGINFO** flag is set in the **sa_flags** field, the **sa_sigaction** field specifies a signal-catching function.

sa_mask specifies a set of signals to be blocked while the signal handler is active. On entry to the signal handler, that set of signals is added to the set of signals already being blocked when the signal is delivered. In addition, the signal that caused the handler to be executed will also be blocked, unless the **SA_NODEFER** flag has been specified. **SIGSTOP** and **SIGKILL** cannot be blocked (the system silently enforces this restriction).

sa_flags specifies a set of flags used to modify the delivery of the signal. It is formed by a logical OR of any of the following values:

SA_ONSTACK If set and the signal is caught, and if the LWP that is chosen to processes a delivered signal has an alternate signal stack declared with **sigaltstack**(2), then it will process the signal on that stack. Otherwise, the signal is delivered on the LWP main stack. Unbound threads (see **thr_create**(3T)) may not have alternate signal stacks.

SA_RESETHAND If set and the signal is caught, the disposition of the signal is reset to **SIG_DFL** and the signal will not be blocked on entry to the signal handler (**SIGILL**, **SIGTRAP**, and **SIGPWR** cannot be automatically reset when delivered; the system silently enforces this restriction).

SA_NODEFER If set and the signal is caught, the signal will not be automatically blocked by the kernel while it is being caught.

SA_RESTART If set and the signal is caught, certain functions that are interrupted by the execution of this signal's handler are transparently restarted by the system; namely, **read**(2) or **write**(2) on slow devices like terminals, **ioctl**(2), **fcntl**(2), **wait**(2), and **waitid**(2). Otherwise, that function returns an **EINTR** error.

SA_SIGINFO If cleared and the signal is caught, *sig* is passed as the only argument to the signal-catching function. If set and the signal is caught, pending signals of type *sig* are reliably queued to the calling process and two additional arguments are passed to the signal-catching function. If the second argument is not equal to **NULL**, it points to a **siginfo_t** structure containing the reason why the signal was generated (see **siginfo**(5)); the third argument points to a **ucontext_t** structure containing the receiving process's context when the signal was delivered (see **ucontext**(5)).

SA_NOCLDWAIT If set and *sig* equals **SIGCHLD**, the system will not create zombie processes when children of the calling process exit. If the calling process subsequently issues a **wait**(2), it blocks until all of the calling process's child processes terminate, and then returns a value of –1 with **errno** set to **ECHILD.**

SA_NOCLDSTOP If set and *sig* equals **SIGCHLD, SIGCHLD** will not be sent to the calling process when its child processes stop or continue.

SA_WAITSIG If set and *sig* equals **SIGWAITING**, then the system will send **SIGWAITING** to the process when all the LWPs in the process are blocked.

RETURN VALUES

On success, **sigaction( )** returns zero. On failure, it returns –1 and sets **errno** to indicate the error. If **sigaction( )** fails, no new signal handler is installed.

ERRORS

sigaction( ) fails if any of the following is true:

EINVAL The value of the *sig* argument is not a valid signal number or is equal to **SIGKILL** or **SIGSTOP.** In addition, if in a multi-threaded process, it is equal to **SIGWAITING, SIGCANCEL,** or **SIGLWP.**

EFAULT *act* or *oact* points to an illegal address.

SEE ALSO

kill(1), **intro**(2), **exit**(2), **kill**(2), **pause**(2), **sigaltstack**(2), **sigprocmask**(2), **sigsend**(2), **sigsuspend**(2), **wait**(2), **signal**(3C), **sigsetops**(3C), **thr_create**(3T), **siginfo**(5), **signal**(5), **ucontext**(5)

NOTES

The handler routine can be declared:

void handler (int sig, siginfo_t *sip, ucontext_t *uap);

Here, **sig** is the signal number. **sip** is a pointer (to space on the stack) to a **siginfo_t** structure, which provides additional detail about the delivery of the signal. **uap** is a pointer (again to space on the stack) to a **ucontext_t** structure (defined in **sys/ucontext.h**) which contains the context from before the signal. It is not recommended that **uap** be used by the handler to restore the context from before the signal delivery.