NAME

sigaction − examine and change a signal action

SYNOPSIS

#include <signal.h>

int sigaction(int *signum***, const struct sigaction ****act***,**
struct sigaction ****oldact*);**

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

sigaction(): _POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _POSIX_SOURCE

DESCRIPTION

The **sigaction**() system call is used to change the action taken by a process on receipt of a specific signal. (See **signal**(7) for an overview of signals.)

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non-null, the new action for signal *signum* is installed from *act*. If *oldact* is non-null, the previous action is saved in *oldact*.

The *sigaction* structure is defined as something like:

```
struct sigaction {
   void     (*sa_handler)(int);
   void     (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t   sa_mask;
   int        sa_flags;
   void     (*sa_restorer)(void);
};
```

On some architectures a union is involved: do not assign to both *sa_handler* and *sa_sigaction*.

The *sa_restorer* element is obsolete and should not be used. POSIX does not specify a *sa_restorer* element.

sa_handler specifies the action to be associated with *signum* and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.

If **SA_SIGINFO** is specified in *sa_flags*, then *sa_sigaction* (instead of *sa_handler*) specifies the signal-handling function for *signum*. This function receives the signal number as its first argument, a pointer to a *siginfo_t* as its second argument and a pointer to a *ucontext_t* (cast to *void **) as its third argument.

sa_mask specifies a mask of signals which should be blocked (i.e., added to the signal mask of the thread in which the signal handler is invoked) during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA_NODEFER** flag is used.

sa_flags specifies a set of flags which modify the behavior of the signal. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP
If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when they receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN** or **SIGTTOU**) or resume (i.e., they receive **SIGCONT**) (see **wait**(2)). This flag is only meaningful when establishing a handler for **SIGCHLD**.

SA_NOCLDWAIT (Since Linux 2.6)
If *signum* is **SIGCHLD**, do not transform children into zombies when they terminate. See also **waitpid**(2). This flag is only meaningful when establishing a handler for **SIGCHLD**, or when setting that signal's disposition to **SIG_DFL**.

If the **SA_NOCLDWAIT** flag is set when establishing a handler for **SIGCHLD**, POSIX.1 leaves it unspecified whether a **SIGCHLD** signal is generated when a child process terminates. On Linux, a **SIGCHLD** signal is generated in this case; on some other implementations, it is not.

SA_NODEFER
Do not prevent the signal from being received from within its own signal handler. This flag is only meaningful when establishing a signal handler. **SA_NOMASK** is an obsolete, non-standard synonym for this flag.

SA_ONSTACK
Call the signal handler on an alternate signal stack provided by **sigaltstack**(2). If an alternate stack is not available, the default stack will be used. This flag is only meaningful when establishing a signal handler.

SA_RESETHAND
Restore the signal action to the default state once the signal handler has been called. This flag is only meaningful when establishing a signal handler. **SA_ONESHOT** is an obsolete, non-standard synonym for this flag.

SA_RESTART
Provide behavior compatible with BSD signal semantics by making certain system calls restartable across signals. This flag is only meaningful when establishing a signal handler. See **signal**(7) for a discussion of system call restarting.

SA_SIGINFO (since Linux 2.2)
The signal handler takes 3 arguments, not one. In this case, *sa_sigaction* should be set instead of *sa_handler*. This flag is only meaningful when establishing a signal handler.

The *siginfo_t* argument to *sa_sigaction* is a struct with the following elements:

```
siginfo_t {
  int     si_signo;   /* Signal number */
  int     si_errno;   /* An errno value */
  int     si_code;    /* Signal code */
  int     si_trapno;  /* Trap number that caused
                         hardware-generated signal
                         (unused on most architectures) */
  pid_t   si_pid;     /* Sending process ID */
  uid_t   si_uid;     /* Real user ID of sending process */
  int     si_status;  /* Exit value or signal */
  clock_t si_utime;   /* User time consumed */
  clock_t si_stime;   /* System time consumed */
  sigval_t si_value;  /* Signal value */
  int     si_int;     /* POSIX.1b signal */
  void    *si_ptr;    /* POSIX.1b signal */
  int     si_overrun; /* Timer overrun count; POSIX.1b timers */
  int     si_timerid; /* Timer ID; POSIX.1b timers */
  void    *si_addr;   /* Memory location which caused fault */
  int     si_band;    /* Band event */
  int     si_fd;      /* File descriptor */
}
```

si_signo, *si_errno* and *si_code* are defined for all signals. (*si_errno* is generally unused on Linux.) The rest of the struct may be a union, so that one should only read the fields that are meaningful for the given signal:

* POSIX.1b signals and **SIGCHLD** fill in *si_pid* and *si_uid*.

* POSIX.1b timers (since Linux 2.6) fill in *si_overrun* and *si_timerid*. The *si_timerid* field is an internal ID used by the kernel to identify the timer; it is not the same as the timer ID returned by **timer_create**(2).
* **SIGCHLD** fills in *si_status*, *si_utime* and *si_stime*. The *si_utime* and *si_stime* fields do not include the times used by waited for children (unlike **getrusage**(2) and **time**(2). In kernels up to 2.6, and since 2.6.27, these fields report CPU time in units of *sysconf(_SC_CLK_TCK)*. In 2.6 kernels before 2.6.27, a bug meant that these fields reported time in units of the (configurable) system jiffy (see **time**(7)).

* *si_int* and *si_ptr* are specified by the sender of the POSIX.1b signal. See **sigqueue**(2) for more details.
* **SIGILL**, **SIGFPE**, **SIGSEGV**, and **SIGBUS** fill in *si_addr* with the address of the fault. **SIGPOLL** fills in *si_band* and *si_fd*.

si_code is a value (not a bit mask) indicating why this signal was sent. The following list shows the values which can be placed in *si_code* for any signal, along with reason that the signal was generated.

SI_USER	**kill**(2) or **raise**(3)
SI_KERNEL	Sent by the kernel.
SI_QUEUE	**sigqueue**(2)
SI_TIMER	POSIX timer expired
SI_MESGQ	POSIX message queue state changed (since Linux 2.6.6); see **mq_notify**(3)
SI_ASYNCIO	AIO completed
SI_SIGIO	queued SIGIO
SI_TKILL	**tkill**(2) or **tgkill**(2) (since Linux 2.4.19)

The following values can be placed in *si_code* for a **SIGILL** signal:

ILL_ILLOPC	illegal opcode
ILL_ILLOPN	illegal operand
ILL_ILLADR	illegal addressing mode
ILL_ILLTRP	illegal trap
ILL_PRVOPC	privileged opcode
ILL_PRVREG	privileged register
ILL_COPROC	coprocessor error
ILL_BADSTK	internal stack error

The following values can be placed in *si_code* for a **SIGFPE** signal:

FPE_INTDIV	integer divide by zero
FPE_INTOVF	integer overflow
FPE_FLTDIV	floating-point divide by zero
FPE_FLTOVF	floating-point overflow
FPE_FLTUND	floating-point underflow
FPE_FLTRES	floating-point inexact result
FPE_FLTINV	floating-point invalid operation
FPE_FLTSUB	subscript out of range

The following values can be placed in *si_code* for a **SIGSEGV** signal:

SEGV_MAPERR
address not mapped to object

SEGV_ACCERR
invalid permissions for mapped object

The following values can be placed in *si_code* for a **SIGBUS** signal:

BUS_ADRALN invalid address alignment

BUS_ADRERR nonexistent physical address

BUS_OBJERR object-specific hardware error

The following values can be placed in *si_code* for a **SIGTRAP** signal:

TRAP_BRKPT process breakpoint

TRAP_TRACE process trace trap

The following values can be placed in *si_code* for a **SIGCHLD** signal:

CLD_EXITED child has exited

CLD_KILLED child was killed

CLD_DUMPED child terminated abnormally

CLD_TRAPPED
traced child has trapped

CLD_STOPPED
child has stopped

CLD_CONTINUED
stopped child has continued (since Linux 2.6.9)

The following values can be placed in *si_code* for a **SIGPOLL** signal:

POLL_IN data input available

POLL_OUT output buffers available

POLL_MSG input message available

POLL_ERR i/o error

POLL_PRI high priority input available

POLL_HUP device disconnected

RETURN VALUE

sigaction() returns 0 on success and −1 on error.

ERRORS

EFAULT
act or *oldact* points to memory which is not a valid part of the process address space.

EINVAL
An invalid signal was specified. This will also be generated if an attempt is made to change the action for **SIGKILL** or **SIGSTOP**, which cannot be caught or ignored.

CONFORMING TO

POSIX.1-2001, SVr4.

NOTES

A child created via **fork**(2) inherits a copy of its parent's signal dispositions. During an **execve**(2), the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

According to POSIX, the behavior of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by **kill**(2) or **raise**(3). Integer division by zero has undefined

result. On some architectures it will generate a **SIGFPE** signal. (Also dividing the most negative integer by −1 may generate **SIGFPE**.) Ignoring this signal might lead to an endless loop.

POSIX.1-1990 disallowed setting the action for **SIGCHLD** to **SIG_IGN**. POSIX.1-2001 allows this possibility, so that ignoring **SIGCHLD** can be used to prevent the creation of zombies (see **wait**(2)). Nevertheless, the historical BSD and System V behaviors for ignoring **SIGCHLD** differ, so that the only completely portable method of ensuring that terminated children do not become zombies is to catch the **SIGCHLD** signal and perform a **wait**(2) or similar.

POSIX.1-1990 only specified **SA_NOCLDSTOP**. POSIX.1-2001 added **SA_NOCLDWAIT**, **SA_RESETHAND**, **SA_NODEFER**, and **SA_SIGINFO**. Use of these latter values in *sa_flags* may be less portable in applications intended for older Unix implementations.

The **SA_RESETHAND** flag is compatible with the SVr4 flag of the same name.

The **SA_NODEFER** flag is compatible with the SVr4 flag of the same name under kernels 1.3.9 and newer. On older kernels the Linux implementation allowed the receipt of any signal, not just the one we are installing (effectively overriding any *sa_mask* settings).

sigaction() can be called with a null second argument to query the current signal handler. It can also be used to check whether a given signal is valid for the current machine by calling it with null second and third arguments.

It is not possible to block **SIGKILL** or **SIGSTOP** (by specifying them in *sa_mask*). Attempts to do so are silently ignored.

See **sigsetops**(3) for details on manipulating signal sets.

See **signal**(7) for a list of the async-signal-safe functions that can be safely called inside from inside a signal handler.

Undocumented

Before the introduction of **SA_SIGINFO** it was also possible to get some additional information, namely by using a *sa_handler* with second argument of type *struct sigcontext*. See the relevant kernel sources for details. This use is obsolete now.

BUGS

In kernels up to and including 2.6.13, specifying **SA_NODEFER** in *sa_flags* prevents not only the delivered signal from being masked during execution of the handler, but also the signals specified in *sa_mask*. This bug was fixed in kernel 2.6.14.

EXAMPLE

See **mprotect**(2).

SEE ALSO

kill(1), **kill**(2), **killpg**(2), **pause**(2), **sigaltstack**(2), **signal**(2), **signalfd**(2), **sigpending**(2), **sigprocmask**(2), **sigqueue**(2), **sigsuspend**(2), **wait**(2), **raise**(3), **siginterrupt**(3), **sigsetops**(3), **sigvec**(3), **core**(5), **signal**(7)

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.