NAME | SYNOPSIS | DESCRIPTION | RETURN VALUE | ERRORS | CONFORMING TO | NOTES | SEE ALSO | COLOPHON

Search online pages

NAME top

signal - ANSI C signal handling

SYNOPSIS top

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION top

The behavior of **signal**() varies across UNIX versions, and has also varied historically across different versions of Linux. **Avoid its use:** use sigaction(2) instead. See *Portability* below.

signal() sets the disposition of the signal *signum* to *handler*, which is either **SIG_IGN**, **SIG_DFL**, or the address of a programmer-defined function (a "signal handler").

If the signal *signum* is delivered to the process, then one of the following happens:

* If the disposition is set to **SIG_IGN**, then the signal is ignored.

* If the disposition is set to **SIG_DFL**, then the default action associated with the signal (see signal(7)) occurs.

* If the disposition is set to a function, then first either the disposition is reset to **SIG_DFL**, or the signal is blocked (see *Portability* below), and then *handler* is called with argument *signum*. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.

The signals **SIGKILL** and **SIGSTOP** cannot be caught or ignored.

RETURN VALUE top

signal() returns the previous value of the signal handler, or **SIG_ERR** on error. In the event of an error, *errno* is set to indicate the cause.

ERRORS top

EINVAL *signum* is invalid.

CONFORMING TO top

POSIX.1-2001, POSIX.1-2008, C89, C99.

NOTES top

The effects of **signal**() in a multithreaded process are unspecified.

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.

See sigaction(2) for details on what happens when **SIGCHLD** is set to **SIG_IGN**.

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

The use of *sighandler_t* is a GNU extension, exposed if **_GNU_SOURCE** is defined; glibc also defines (the BSD-derived) *sig_t* if **_BSD_SOURCE** (glibc 2.19 and earlier) or **_DEFAULT_SOURCE** (glibc 2.19 and later) is defined. Without use of such a type, the declaration of **signal**() is the somewhat harder to read:

```
void ( *signal(int signum, void (*handler)(int)) ) (int);
```

Portability

The only portable use of **signal**() is to set a signal's disposition to **SIG_DFL** or **SIG_IGN**. The semantics when using **signal**() to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); **do not use it for this purpose.**

POSIX.1 solved the portability mess by specifying sigaction(2), which provides explicit control of the semantics when a signal handler is invoked; use that interface instead of **signal**().

In the original UNIX systems, when a handler that was established using **signal**() was invoked by the delivery of a signal, the disposition of the signal would be reset to **SIG_DFL**, and the system did not block delivery of further instances of the signal. This is

equivalent to calling sigaction(2) with the following flags:

```
sa.sa_flags = SA_RESETHAND | SA_NODEFER;
```

System V also provides these semantics for **signal**(). This was bad because the signal might be delivered again before the handler had a chance to reestablish itself. Furthermore, rapid deliveries of the same signal could result in recursive invocations of the handler.

BSD improved on this situation, but unfortunately also changed the semantics of the existing **signal**() interface while doing so. On BSD, when a signal handler is invoked, the signal disposition is not reset, and further instances of the signal are blocked from being delivered while the handler is executing. Furthermore, certain blocking system calls are automatically restarted if interrupted by a signal handler (see signal(7)). The BSD semantics are equivalent to calling sigaction(2) with the following flags:

```
sa.sa_flags = SA_RESTART;
```

The situation on Linux is as follows:

* The kernel's **signal**() system call provides System V semantics.

* By default, in glibc 2 and later, the **signal**() wrapper function does not invoke the kernel system call. Instead, it calls sigaction(2) using flags that supply BSD semantics. This default behavior is provided as long as a suitable feature test macro is defined: **_BSD_SOURCE** on glibc 2.19 and earlier or **_DEFAULT_SOURCE** in glibc 2.19 and later. (By default, these macros are defined; see feature_test_macros(7) for details.) If such a feature test macro is not defined, then **signal**() provides System V semantics.

SEE ALSO top

kill(1), alarm(2), kill(2), killpg(2), pause(2), sigaction(2), signalfd(2), sigpending(2), sigprocmask(2), sigsuspend(2), bsd_signal(3), raise(3), siginterrupt(3), sigqueue(3), sigsetops(3), sigvec(3), sysv_signal(3), signal(7)

COLOPHON top

This page is part of release 4.08 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

HTML rendering created 2016-10-08 by Michael Kerrisk, author of *The Linux Programming Interface*, maintainer of the Linux *man-pages* project.

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look here.

Hosting by jambit GmbH.