NAME
signal – overview of signals

DESCRIPTION
Linux supports both POSIX reliable signals (hereinafter "standard signals") and POSIX real-time signals.

Signal dispositions

Each signal has a current *disposition*, which determines how the process behaves when it is delivered the signal.

The entries in the "Action" column of the table below specify the default disposition for each signal, as follows:

Term   Default action is to terminate the process.

Ign   Default action is to ignore the signal.

Core   Default action is to terminate the process and dump core (see **core**(5)).

Stop   Default action is to stop the process.

Cont   Default action is to continue the process if it is currently stopped.

A process can change the disposition of a signal using **sigaction**(2) or **signal**(2). (The latter is less portable when establishing a signal handler; see **signal**(2) for details.) Using these system calls, a process can elect one of the following behaviors to occur on delivery of the signal: perform the default action; ignore the signal; or catch the signal with a *signal handler*, a programmer-defined function that is automatically invoked when the signal is delivered.

By default, a signal handler is invoked on the normal process stack. It is possible to arrange that the signal handler uses an alternate stack; see **sigaltstack**(2) for a discussion of how to do this and when it might be useful.

The signal disposition is a per-process attribute: in a multithreaded application, the disposition of a particular signal is the same for all threads.

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.

Sending a signal

The following system calls and library functions allow the caller to send a signal:

raise(3)   Sends a signal to the calling thread.

kill(2)   Sends a signal to a specified process, to all members of a specified process group, or to all processes on the system.

killpg(3)   Sends a signal to all of the members of a specified process group.

pthread_kill(3)   Sends a signal to a specified POSIX thread in the same process as the caller.

tgkill(2)   Sends a signal to a specified thread within a specific process. (This is the system call used to implement **pthread_kill**(3).)

sigqueue(3)   Sends a real-time signal with accompanying data to a specified process.

Waiting for a signal to be caught

The following system calls suspend execution of the calling thread until a signal is caught (or an unhandled signal terminates the process):

pause(2)   Suspends execution until any signal is caught.

sigsuspend(2)   Temporarily changes the signal mask (see below) and suspends execution until one of the unmasked signals is caught.

Synchronously accepting a signal

Rather than asynchronously catching a signal via a signal handler, it is possible to synchronously accept the signal, that is, to block execution until the signal is delivered, at which point the kernel returns information

about the signal to the caller. There are two general ways to do this:

* **sigwaitinfo**(2), **sigtimedwait**(2), and **sigwait**(3) suspend execution until one of the signals in a specified set is delivered. Each of these calls returns information about the delivered signal.

* **signalfd**(2) returns a file descriptor that can be used to read information about signals that are delivered to the caller. Each **read**(2) from this file descriptor blocks until one of the signals in the set specified in the **signalfd**(2) call is delivered to the caller. The buffer returned by **read**(2) contains a structure describing the signal.

Signal mask and pending signals

A signal may be *blocked*, which means that it will not be delivered until it is later unblocked. Between the time when it is generated and when it is delivered a signal is said to be *pending*.

Each thread in a process has an independent *signal mask*, which indicates the set of signals that the thread is currently blocking. A thread can manipulate its signal mask using **pthread_sigmask**(3). In a traditional single-threaded application, **sigprocmask**(2) can be used to manipulate the signal mask.

A child created via **fork**(2) inherits a copy of its parent's signal mask; the signal mask is preserved across **execve**(2).

A signal may be process-directed or thread-directed. A process-directed signal is one that is targeted at (and thus pending for) the process as a whole. A signal may be process-directed because it was generated by the kernel for reasons other than a hardware exception, or because it was sent using **kill**(2) or **sigqueue**(3). A thread-directed signal is one that is targeted at a specific thread. A signal may be thread-directed because it was generated as a consequence of executing a specific machine-language instruction that triggered a hardware exception (e.g., **SIGSEGV** for an invalid memory access, or **SIGFPE** for a math error), or because it was it was targeted at a specific thread using interfaces such as **tgkill**(2) or **pthread_kill**(3).

A process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked. If more than one of the threads has the signal unblocked, then the kernel chooses an arbitrary thread to which to deliver the signal.

A thread can obtain the set of signals that it currently has pending using **sigpending**(2). This set will consist of the union of the set of pending process-directed signals and the set of signals pending for the calling thread.

A child created via **fork**(2) initially has an empty pending signal set; the pending signal set is preserved across an **execve**(2).

Standard signals

Linux supports the standard signals listed below. The second column of the table indicates which standard (if any) specified the signal: "P1990" indicates that the signal is described in the original POSIX.1-1990 standard; "P2001" indicates that the signal was added in SUSv2 and POSIX.1-2001.

Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from **abort**(3)
SIGALRM	P1990	Term	Timer signal from **alarm**(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	–	Ign	A synonym for **SIGCHLD**
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	–	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	–		A synonym for **SIGPWR**
SIGINT	P1990	Term	Interrupt from keyboard

SIGIO	–	Term	I/O now possible (4.2BSD)
SIGIOT	–	Core	IOT trap. A synonym for **SIGABRT**
SIGKILL	P1990	Term	Kill signal
SIGLOST	–	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see **pipe**(7)
SIGPOLL	P2001	Term	Pollable event (Sys V). Synonym for **SIGIO**
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	–	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	–	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also **seccomp**(2)
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	–	Core	Synonymous with **SIGSYS**
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD); see **setrlimit**(2)
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD); see **setrlimit**(2)
SIGWINCH	–	Ign	Window resize signal (4.3BSD, Sun)

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

Up to and including Linux 2.2, the default behavior for **SIGSYS**, **SIGXCPU**, **SIGXFSZ**, and (on architectures other than SPARC and MIPS) **SIGBUS** was to terminate the process (without a core dump). (On some other UNIX systems the default action for **SIGXCPU** and **SIGXFSZ** is to terminate the process without a core dump.) Linux 2.4 conforms to the POSIX.1-2001 requirements for these signals, terminating the process with a core dump.

SIGEMT is not specified in POSIX.1-2001, but nevertheless appears on most other UNIX systems, where its default action is typically to terminate the process with a core dump.

SIGPWR (which is not specified in POSIX.1-2001) is typically ignored by default on those other UNIX systems where it appears.

SIGIO (which is not specified in POSIX.1-2001) is ignored by default on several other UNIX systems.

Queueing and delivery semantics for standard signals

If multiple standard signals are pending for a process, the order in which the signals are delivered is unspecified.

Standard signals do not queue. If multiple instances of a standard signal are generated while that signal is blocked, then only one instance of the signal is marked as pending (and the signal will be delivered just once when it is unblocked). In the case where a standard signal is already pending, the *siginfo_t* structure (see **sigaction**(2)) associated with that signal is not overwritten on arrival of subsequent instances of the same signal. Thus, the process will receive the information associated with the first instance of the signal.

Signal numbering for standard signals

The numeric value for each signal is given in the table below. As shown in the table, many signals have different numeric values on different architectures. The first numeric value in each table row shows the signal number on x86, ARM, and most other architectures; the second value is for Alpha and SPARC; the third is for MIPS; and the last is for PARISC. A dash (−) denotes that a signal is absent on the corresponding architecture.

Signal	x86/ARM most others	Alpha/ SPARC	MIPS	PARISC	Notes
SIGHUP	1	1	1	1	
SIGINT	2	2	2	2	
SIGQUIT	3	3	3	3	
SIGILL	4	4	4	4	
SIGTRAP	5	5	5	5	
SIGABRT	6	6	6	6	
SIGIOT	6	6	6	6	
SIGBUS	7	10	10	10	
SIGEMT	−	7	7	-	
SIGFPE	8	8	8	8	
SIGKILL	9	9	9	9	
SIGUSR1	10	30	16	16	
SIGSEGV	11	11	11	11	
SIGUSR2	12	31	17	17	
SIGPIPE	13	13	13	13	
SIGALRM	14	14	14	14	
SIGTERM	15	15	15	15	
SIGSTKFLT	16	−	−	7	
SIGCHLD	17	20	18	18	
SIGCLD	−	−	18	−	
SIGCONT	18	19	25	26	
SIGSTOP	19	17	23	24	
SIGTSTP	20	18	24	25	
SIGTTIN	21	21	26	27	
SIGTTOU	22	22	27	28	
SIGURG	23	16	21	29	
SIGXCPU	24	24	30	12	
SIGXFSZ	25	25	31	30	
SIGVTALRM	26	26	28	20	
SIGPROF	27	27	29	21	
SIGWINCH	28	28	20	23	
SIGIO	29	23	22	22	
SIGPOLL					Same as SIGIO
SIGPWR	30	29/−	19	19	
SIGINFO	−	29/−	−	−	
SIGLOST	−	−/29	−	−	
SIGSYS	31	12	12	31	
SIGUNUSED	31	−	−	31	

Note the following:

* Where defined, **SIGUNUSED** is synonymous with **SIGSYS**. Since glibc 2.26, **SIGUNUSED** is no longer defined on any architecture.

* Signal 29 is **SIGINFO/SIGPWR** (synonyms for the same value) on Alpha but **SIGLOST** on SPARC.

Real-time signals

Starting with version 2.2, Linux supports real-time signals as originally defined in the POSIX.1b real-time extensions (and now included in POSIX.1-2001). The range of supported real-time signals is defined by the macros **SIGRTMIN** and **SIGRTMAX**. POSIX.1-2001 requires that an implementation support at least **_POSIX_RTSIG_MAX** (8) real-time signals.

The Linux kernel supports a range of 33 different real-time signals, numbered 32 to 64. However, the glibc POSIX threads implementation internally uses two (for NPTL) or three (for LinuxThreads) real-time signals (see **pthreads**(7)), and adjusts the value of **SIGRTMIN** suitably (to 34 or 35). Because the range of available real-time signals varies according to the glibc threading implementation (and this variation can occur at run time according to the available kernel and glibc), and indeed the range of real-time signals varies across UNIX systems, programs should *never refer to real-time signals using hard-coded numbers*, but instead should always refer to real-time signals using the notation **SIGRTMIN**+n, and include suitable (run-time) checks that **SIGRTMIN**+n does not exceed **SIGRTMAX**.

Unlike standard signals, real-time signals have no predefined meanings: the entire set of real-time signals can be used for application-defined purposes.

The default action for an unhandled real-time signal is to terminate the receiving process.

Real-time signals are distinguished by the following:

1. Multiple instances of real-time signals can be queued. By contrast, if multiple instances of a standard signal are delivered while that signal is currently blocked, then only one instance is queued.

2. If the signal is sent using **sigqueue**(3), an accompanying value (either an integer or a pointer) can be sent with the signal. If the receiving process establishes a handler for this signal using the **SA_SIG-INFO** flag to **sigaction**(2), then it can obtain this data via the *si_value* field of the *siginfo_t* structure passed as the second argument to the handler. Furthermore, the *si_pid* and *si_uid* fields of this structure can be used to obtain the PID and real user ID of the process sending the signal.

3. Real-time signals are delivered in a guaranteed order. Multiple real-time signals of the same type are delivered in the order they were sent. If different real-time signals are sent to a process, they are delivered starting with the lowest-numbered signal. (I.e., low-numbered signals have highest priority.) By contrast, if multiple standard signals are pending for a process, the order in which they are delivered is unspecified.

If both standard and real-time signals are pending for a process, POSIX leaves it unspecified which is delivered first. Linux, like many other implementations, gives priority to standard signals in this case.

According to POSIX, an implementation should permit at least **_POSIX_SIGQUEUE_MAX** (32) real-time signals to be queued to a process. However, Linux does things differently. In kernels up to and including 2.6.7, Linux imposes a system-wide limit on the number of queued real-time signals for all processes. This limit can be viewed and (with privilege) changed via the */proc/sys/kernel/rtsig-max* file. A related file, */proc/sys/kernel/rtsig-nr*, can be used to find out how many real-time signals are currently queued. In Linux 2.6.8, these */proc* interfaces were replaced by the **RLIMIT_SIGPENDING** resource limit, which specifies a per-user limit for queued signals; see **setrlimit**(2) for further details.

The addition of real-time signals required the widening of the signal set structure (*sigset_t*) from 32 to 64 bits. Consequently, various system calls were superseded by new system calls that supported the larger signal sets. The old and new system calls are as follows:

Linux 2.0 and earlier	Linux 2.2 and later
sigaction(2)	**rt_sigaction**(2)
sigpending(2)	**rt_sigpending**(2)
sigprocmask(2)	**rt_sigprocmask**(2)
sigreturn(2)	**rt_sigreturn**(2)
sigsuspend(2)	**rt_sigsuspend**(2)
sigtimedwait(2)	**rt_sigtimedwait**(2)

Interruption of system calls and library functions by signal handlers

If a signal handler is invoked while a system call or library function call is blocked, then either:

* the call is automatically restarted after the signal handler returns; or

* the call fails with the error **EINTR**.

Which of these two behaviors occurs depends on the interface and whether or not the signal handler was established using the **SA_RESTART** flag (see **sigaction**(2)). The details vary across UNIX systems; below, the details for Linux.

If a blocked call to one of the following interfaces is interrupted by a signal handler, then the call is automatically restarted after the signal handler returns if the **SA_RESTART** flag was used; otherwise the call fails with the error **EINTR**:

* **read**(2), **readv**(2), **write**(2), **writev**(2), and **ioctl**(2) calls on "slow" devices. A "slow" device is one where the I/O call may block for an indefinite time, for example, a terminal, pipe, or socket. If an I/O call on a slow device has already transferred some data by the time it is interrupted by a signal handler, then the call will return a success status (normally, the number of bytes transferred). Note that a (local) disk is not a slow device according to this definition; I/O operations on disk devices are not interrupted by signals.

* **open**(2), if it can block (e.g., when opening a FIFO; see **fifo**(7)).

* **wait**(2), **wait3**(2), **wait4**(2), **waitid**(2), and **waitpid**(2).

* Socket interfaces: **accept**(2), **connect**(2), **recv**(2), **recvfrom**(2), **recvmmsg**(2), **recvmsg**(2), **send**(2), **sendto**(2), and **sendmsg**(2), unless a timeout has been set on the socket (see below).

* File locking interfaces: **flock**(2) and the **F_SETLKW** and **F_OFD_SETLKW** operations of **fcntl**(2)

* POSIX message queue interfaces: **mq_receive**(3), **mq_timedreceive**(3), **mq_send**(3), and **mq_timedsend**(3).

* **futex**(2) **FUTEX_WAIT** (since Linux 2.6.22; beforehand, always failed with **EINTR**).

* **getrandom**(2).

* **pthread_mutex_lock**(3), **pthread_cond_wait**(3), and related APIs.

* **futex**(2) **FUTEX_WAIT_BITSET**.

* POSIX semaphore interfaces: **sem_wait**(3) and **sem_timedwait**(3) (since Linux 2.6.22; beforehand, always failed with **EINTR**).

* **read**(2) from an **inotify**(7) file descriptor (since Linux 3.8; beforehand, always failed with **EINTR**).

The following interfaces are never restarted after being interrupted by a signal handler, regardless of the use of **SA_RESTART**; they always fail with the error **EINTR** when interrupted by a signal handler:

* "Input" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using **setsockopt**(2): **accept**(2), **recv**(2), **recvfrom**(2), **recvmmsg**(2) (also with a non-NULL *timeout* argument), and **recvmsg**(2).

* "Output" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using **setsockopt**(2): **connect**(2), **send**(2), **sendto**(2), and **sendmsg**(2).

* Interfaces used to wait for signals: **pause**(2), **sigsuspend**(2), **sigtimedwait**(2), and **sigwaitinfo**(2).

* File descriptor multiplexing interfaces: **epoll_wait**(2), **epoll_pwait**(2), **poll**(2), **ppoll**(2), **select**(2), and **pselect**(2).

* System V IPC interfaces: **msgrcv**(2), **msgsnd**(2), **semop**(2), and **semtimedop**(2).

* Sleep interfaces: **clock_nanosleep**(2), **nanosleep**(2), and **usleep**(3).

* **io_getevents**(2).

The **sleep**(3) function is also never restarted if interrupted by a handler, but gives a success return: the

number of seconds remaining to sleep.

Interruption of system calls and library functions by stop signals

On Linux, even in the absence of signal handlers, certain blocking interfaces can fail with the error **EINTR** after the process is stopped by one of the stop signals and then resumed via **SIGCONT**. This behavior is not sanctioned by POSIX.1, and doesn't occur on other systems.

The Linux interfaces that display this behavior are:

* "Input" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using **setsock-opt**(2): **accept**(2), **recv**(2), **recvfrom**(2), **recvmmsg**(2) (also with a non-NULL *timeout* argument), and **recvmsg**(2).

* "Output" socket interfaces, when a timeout (**SO_RCVTIMEO**) has been set on the socket using **setsock-opt**(2): **connect**(2), **send**(2), **sendto**(2), and **sendmsg**(2), if a send timeout (**SO_SNDTIMEO**) has been set.

* **epoll_wait**(2), **epoll_pwait**(2).

* **semop**(2), **semtimedop**(2).

* **sigtimedwait**(2), **sigwaitinfo**(2).

* Linux 3.7 and earlier: **read**(2) from an **inotify**(7) file descriptor

* Linux 2.6.21 and earlier: **futex**(2) **FUTEX_WAIT**, **sem_timedwait**(3), **sem_wait**(3).

* Linux 2.6.8 and earlier: **msgrcv**(2), **msgsnd**(2).

* Linux 2.4 and earlier: **nanosleep**(2).

CONFORMING TO

POSIX.1, except as noted.

NOTES

For a discussion of async-signal-safe functions, see **signal-safety**(7).

The */proc/[pid]/task/[tid]/status* file contains various fields that show the signals that a thread is blocking (*SigBlk*), catching (*SigCgt*), or ignoring (*SigIgn*). (The set of signals that are caught or ignored will be the same across all threads in a process.) Other fields show the set of pending signals that are directed to the thread (*SigPnd*) as well as the set of pending signals that are directed to the process as a whole (*ShdPnd*). The corresponding fields in */proc/[pid]/status* show the information for the main thread. See **proc**(5) for further details.

SEE ALSO

kill(1), **clone**(2), **getrlimit**(2), **kill**(2), **restart_syscall**(2), **rt_sigqueueinfo**(2), **pidfd_send_signal**(2), **setitimer**(2), **setrlimit**(2), **sgetmask**(2), **sigaction**(2), **sigaltstack**(2), **signal**(2), **signalfd**(2), **sigpending**(2), **sigprocmask**(2), **sigreturn**(2), **sigsuspend**(2), **sigwaitinfo**(2), **abort**(3), **bsd_signal**(3), **killpg**(3), **longjmp**(3), **pthread_sigqueue**(3), **raise**(3), **sigqueue**(3), **sigset**(3), **sigsetops**(3), **sigvec**(3), **sigwait**(3), **strsignal**(3), **sysv_signal**(3), **core**(5), **proc**(5), **nptl**(7), **pthreads**(7), **sigevent**(7)

COLOPHON

This page is part of release 5.05 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/.