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 tables 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 (less portably) **signal**(2). 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, the 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(2) 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(2) 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 process or 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 generated (and thus pending) for a process as a whole (e.g., when sent using **kill**(2)) or for a specific thread (e.g., certain signals, such as **SIGSEGV** and **SIGFPE**, generated as a consequence of executing a specific machine-language instruction are thread directed, as are signals targeted at a specific thread using **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. Several signal numbers are architecture-dependent, as indicated in the "Value" column. (Where three values are given, the first one is usually valid for alpha and sparc, the middle one for ix86, ia64, ppc, s390, arm and sh, and the last one for mips. A − denotes that a signal is absent on the corresponding architecture.)

First the signals described in the original POSIX.1-1990 standard.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from **abort**(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from **alarm**(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1

SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

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

Next the signals not in the POSIX.1-1990 standard but described in SUSv2 and POSIX.1-2001.

Signal	Value	Action	Comment
SIGBUS	10,7,10	Core	Bus error (bad memory access)
SIGPOLL		Term	Pollable event (Sys V). Synonym for **SIGIO**
SIGPROF	27,27,29	Term	Profiling timer expired
SIGSYS	12,−,12	Core	Bad argument to routine (SVr4)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2BSD)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2BSD)

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.

Next various other signals.

Signal	Value	Action	Comment
SIGIOT	6	Core	IOT trap. A synonym for **SIGABRT**
SIGEMT	7,−,7	Term	
SIGSTKFLT	−,16,−	Term	Stack fault on coprocessor (unused)
SIGIO	23,29,22	Term	I/O now possible (4.2BSD)
SIGCLD	−,−,18	Ign	A synonym for **SIGCHLD**
SIGPWR	29,30,19	Term	Power failure (System V)
SIGINFO	29,−,−		A synonym for **SIGPWR**
SIGLOST	−,−,−	Term	File lock lost
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)
SIGUNUSED	−,31,−	Term	Unused signal (will be **SIGSYS**)

(Signal 29 is **SIGINFO** / **SIGPWR** on an alpha but **SIGLOST** on a sparc.)

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.

Real-time Signals

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 32 different real-time signals, numbered 33 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. (Note, however, that the LinuxThreads implementation uses the first three real-time signals.)

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**(2), 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_SIGINFO** 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.

Async-signal-safe functions

A signal handling routine established by **sigaction**(2) or **signal**(2) must be very careful, since processing elsewhere may be interrupted at some arbitrary point in the execution of the program. POSIX has the concept of "safe function". If a signal interrupts the execution of an unsafe function, and *handler* calls an unsafe function, then the behavior of the program is undefined.

POSIX.1-2004 (also known as POSIX.1-2001 Technical Corrigendum 2) requires an implementation to guarantee that the following functions can be safely called inside a signal handler:

_Exit()
_exit()

abort()
accept()
access()
aio_error()
aio_return()
aio_suspend()
alarm()
bind()
cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
chdir()
chmod()
chown()
clock_gettime()
close()
connect()
creat()
dup()
dup2()
execle()
execve()
fchmod()
fchown()
fcntl()
fdatasync()
fork()
fpathconf()
fstat()
fsync()
ftruncate()
getegid()
geteuid()
getgid()
getgroups()
getpeername()
getpgrp()
getpid()
getppid()
getsockname()
getsockopt()
getuid()
kill()
link()
listen()
lseek()
lstat()
mkdir()
mkfifo()
open()
pathconf()
pause()
pipe()

poll()
posix_trace_event()
pselect()
raise()
read()
readlink()
recv()
recvfrom()
recvmsg()
rename()
rmdir()
select()
sem_post()
send()
sendmsg()
sendto()
setgid()
setpgid()
setsid()
setsockopt()
setuid()
shutdown()
sigaction()
sigaddset()
sigdelset()
sigemptyset()
sigfillset()
sigismember()
signal()
sigpause()
sigpending()
sigprocmask()
sigqueue()
sigset()
sigsuspend()
sleep()
sockatmark()
socket()
socketpair()
stat()
symlink()
sysconf()
tcdrain()
tcflow()
tcflush()
tcgetattr()
tcgetpgrp()
tcsendbreak()
tcsetattr()
tcsetpgrp()
time()
timer_getoverrun()
timer_gettime()
timer_settime()

times()
umask()
uname()
unlink()
utime()
wait()
waitpid()
write()

POSIX.1-2008 removes fpathconf(), pathconf(), and sysconf() from the above list, and adds the following functions:

execl()
execv()
faccessat()
fchmodat()
fchownat()
fexecve()
fstatat()
futimens()
linkat()
mkdirat()
mkfifoat()
mknod()
mknodat()
openat()
readlinkat()
renameat()
symlinkat()
unlinkat()
utimensat()
utimes()

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 will be automatically restarted after the signal handler returns if the **SA_RESTART** flag was used; otherwise the call will fail 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. (A disk is not a slow device according to this definition.) 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).

* **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), **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 **fcntl**(2) **F_SETLKW**.
* 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**).
* POSIX semaphore interfaces: **sem_wait**(3) and **sem_timedwait**(3) (since Linux 2.6.22; 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:

* Socket interfaces, when a timeout has been set on the socket using **setsockopt**(2): **accept**(2), **recv**(2), **recvfrom**(2), and **recvmsg**(2), if a receive timeout (**SO_RCVTIMEO**) has been set; **connect**(2), **send**(2), **sendto**(2), and **sendmsg**(2), if a send timeout (**SO_SNDTIMEO**) has been set.
* 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).
* **read**(2) from an **inotify**(7) file descriptor.
* **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:

* Socket interfaces, when a timeout has been set on the socket using **setsockopt**(2): **accept**(2), **recv**(2), **recvfrom**(2), and **recvmsg**(2), if a receive timeout (**SO_RCVTIMEO**) has been set; **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).
* **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.

BUGS

SIGIO and **SIGLOST** have the same value. The latter is commented out in the kernel source, but the build process of some software still thinks that signal 29 is **SIGLOST**.

SEE ALSO

kill(1), **getrlimit**(2), **kill**(2), **killpg**(2), **setitimer**(2), **setrlimit**(2), **sgetmask**(2), **sigaction**(2), **sigaltstack**(2), **signal**(2), **signalfd**(2), **sigpending**(2), **sigprocmask**(2), **sigqueue**(2), **sigsuspend**(2), **sigwaitinfo**(2), **abort**(3), **bsd_signal**(3), **longjmp**(3), **raise**(3), **sigset**(3), **sigsetops**(3), **sigvec**(3), **sigwait**(3),

strsignal(3), **sysv_signal**(3), **core**(5), **proc**(5), **pthreads**(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/.