WIKIPEDIA
The Free Encyclopedia

Signal (IPC)

Signals are standardized messages sent to a running program to trigger specific behavior, such as quitting or error handling. They are a limited form of inter-process communication (IPC), typically used in Unix, Unix-like, and other POSIX-compliant operating systems.

A signal is an asynchronous notification sent to a process or to a specific thread within the same process to notify it of an event. Common uses of signals are to interrupt, suspend, terminate or kill a process. Signals originated in 1970s Bell Labs Unix and were later specified in the POSIX standard.

When a signal is sent, the operating system interrupts the target process's normal flow of execution to deliver the signal. Execution can be interrupted during any non-atomic instruction. If the process has previously registered a **signal handler**, that routine is executed. Otherwise, the default signal handler is executed.

Embedded programs may find signals useful for inter-process communications, as signals are notable for their algorithmic efficiency.

Signals are similar to interrupts, the difference being that interrupts are mediated by the CPU and handled by the kernel while signals are mediated by the kernel (possibly via system calls) and handled by individual processes. The kernel may pass an interrupt as a signal to the process that caused it (typical examples are SIGSEGV, SIGBUS, SIGILL and SIGFPE).

History

- Version 1 Unix (1971) had separate system calls to catch interrupts, quits, and machine traps.
- `kill` appeared in Version 2 (1972).
- Version 4 (1973) combined all traps into one call, `signal`.
- Version 5 (1974) could send arbitrary signals.[1]
- In Version 7 (1979) each numbered trap received a symbolic name.
- Plan 9 from Bell Labs (late 80s) replaced signals with *notes*, which permit sending short, arbitrary strings.[2]

Sending signals

The `kill(2)` system call sends a specified signal to a specified process, if permissions allow. Similarly, the `kill(1)` command allows a user to send signals to processes. The `raise(3)` library function sends the specified signal to the current process.

Exceptions such as division by zero, segmentation violation (SIGSEGV), and floating point exception (SIGFPE) will cause a core dump and terminate the program.

The kernel can generate signals to notify processes of events. For example, SIGPIPE will be generated when a process writes to a pipe which has been closed by the reader; by default, this causes the process to terminate, which is convenient when constructing shell pipelines.

Typing certain key combinations at the controlling terminal of a running process causes the system to send it certain signals:[3]

- Ctrl-C (in older Unixes, DEL) sends an INT signal ("interrupt", SIGINT); by default, this causes the process to terminate.
- Ctrl-Z sends a TSTP signal ("terminal stop", SIGTSTP); by default, this causes the process to suspend execution.[4]
- Ctrl-\ sends a QUIT signal (SIGQUIT); by default, this causes the process to terminate and dump core.
- Ctrl-T (not supported on all UNIXes) sends an INFO signal (SIGINFO); by default, and if supported by the command, this causes the operating system to show information about the running command.[5]

These default key combinations with modern operating systems can be changed with the `stty` command.

Handling signals

Signal handlers can be installed with the `signal(2)` or `sigaction(2)` system call. If a signal handler is not installed for a particular signal, the default handler is used. Otherwise the signal is intercepted and the signal handler is invoked. The process can also specify two default behaviors, without creating a handler: ignore the signal (SIG_IGN) and use the default signal handler (SIG_DFL). There are two signals which cannot be intercepted and handled: SIGKILL and SIGSTOP.

Risks

Signal handling is vulnerable to race conditions. As signals are asynchronous, another signal (even of the same type) can be delivered to the process during execution of the signal handling routine.

The `sigprocmask(2)` call can be used to block and unblock delivery of signals. Blocked signals are not delivered to the process until unblocked. Signals that cannot be ignored (SIGKILL and SIGSTOP) cannot be blocked.

Signals can cause the interruption of a system call in progress, leaving it to the application to manage a non-transparent restart.

Signal handlers should be written in a way that does not result in any unwanted side-effects, e.g. `errno` alteration, signal mask alteration, signal disposition change, and other global process attribute changes. Use of non-reentrant functions, e.g., `malloc` or `printf`, inside signal handlers is also unsafe. In particular, the POSIX specification and the Linux man page `signal (7)` require that all system functions directly or *indirectly* called from a signal function are *async-signal safe*.[6][7] The `signal-safety(7)` man page gives a list of such async-signal safe system functions (practically the system calls), otherwise it is an undefined behavior.[8] It is suggested to simply set some `volatile sig_atomic_t` variable in a signal handler, and to test it elsewhere.[9]

Signal handlers can instead put the signal into a queue and immediately return. The main thread will then continue "uninterrupted" until signals are taken from the queue, such as in an event loop. "Uninterrupted" here means that operations that block may return prematurely and must be resumed, as mentioned above. Signals should be processed from the queue on the main thread and not by worker pools, as that reintroduces the problem of asynchronicity. However, managing a queue is not possible in an async-signal safe way with only `sig_atomic_t`, as only single reads and writes to such variables are guaranteed to be atomic, not increments or (fetch-and)-decrements, as would be required for a queue. Thus, effectively, only one signal per handler can be queued safely with `sig_atomic_t` until it has been processed.

Relationship with hardware exceptions

A process's execution may result in the generation of a hardware exception, for instance, if the process attempts to divide by zero or incurs a page fault.

In Unix-like operating systems, this event automatically changes the processor context to start executing a kernel exception handler. In case of some exceptions, such as a page fault, the kernel has sufficient information to fully handle the event itself and resume the process's execution.

Other exceptions, however, the kernel cannot process intelligently and it must instead defer the exception handling operation to the faulting process. This deferral is achieved via the signal mechanism, wherein the kernel sends to the process a signal corresponding to the current exception. For example, if a process attempted integer divide by zero on an x86 CPU, a *divide error* exception would be generated and cause the kernel to send the SIGFPE signal to the process.

Similarly, if the process attempted to access a memory address outside of its virtual address space, the kernel would notify the process of this violation via a SIGSEGV (segmentation violation signal). The exact mapping between signal names and exceptions is obviously dependent upon the CPU, since exception types differ between architectures.

POSIX signals

The list below documents the signals specified in the Single Unix Specification Version 5. All signals are defined as macro constants in the `<signal.h>` header file. The name of the macro constant consists of a "SIG" prefix followed by a mnemonic name for the signal.

A process can define how to handle incoming POSIX signals. If a process does not define a behaviour for a signal, then the *default handler* for that signal is being used. The table below lists some default actions for POSIX-compliant UNIX systems, such as FreeBSD, OpenBSD and Linux.

Signal	Portable number	Default action	Description
SIGABRT	6	Terminate (core dump)	Process abort signal
SIGALRM	14	Terminate	Alarm clock
SIGBUS	—	Terminate (core dump)	Access to an undefined portion of a memory object
SIGCHLD	—	Ignore	Child process terminated, stopped, or continued
SIGCONT	—	Continue	Continue executing, if stopped
SIGFPE	8	Terminate (core dump)	Erroneous arithmetic operation
SIGHUP	1	Terminate	Hangup
SIGILL	4	Terminate (core dump)	Illegal instruction
SIGINT	2	Terminate	Terminal interrupt signal
SIGKILL	9	Terminate	Kill (cannot be caught or ignored)
SIGPIPE	13	Terminate	Write on a pipe with no one to read it
SIGQUIT	3	Terminate (core dump)	Terminal quit signal
SIGSEGV	11	Terminate (core dump)	Invalid memory reference
SIGSTOP	—	Stop	Stop executing (cannot be caught or ignored)
SIGSYS	—	Terminate (core dump)	Bad system call
SIGTERM	15	Terminate	Termination signal
SIGTRAP	5	Terminate (core dump)	Trace/breakpoint trap
SIGTSTP	—	Stop	Terminal stop signal
SIGTTIN	—	Stop	Background process attempting read
SIGTTOU	—	Stop	Background process attempting write
SIGUSR1	—	Terminate	User-defined signal 1
SIGUSR2	—	Terminate	User-defined signal 2
SIGURG	—	Ignore	Out-of-band data is available at a socket
SIGVTALRM	—	Terminate	Virtual timer expired
SIGXCPU	—	Terminate (core dump)	CPU time limit exceeded
SIGXFSZ	—	Terminate (core dump)	File size limit exceeded
SIGWINCH	—	Ignore	Terminal window size changed

Portable number:

For most signals the corresponding signal number is implementation-defined. This column lists the numbers specified in the POSIX standard.[10]

Actions explained:

Terminate – Abnormal termination of the process. The process is terminated with all the consequences of _exit() except that the status made available to wait() and waitpid() indicates abnormal termination by the specified signal.
Terminate (core dump) – Abnormal termination of the process. Additionally, implementation-defined abnormal termination actions, such as creation of a core file, may occur.
Ignore – Ignore the signal.

Stop – Stop (or suspend) the process.
Continue – Continue the process, if it is stopped; otherwise, ignore the signal.

`SIGABRT` and `SIGIOT`

The SIGABRT signal is sent to a process to tell it to **abort**, i.e. to terminate. The signal is usually initiated by the process itself when it calls `abort()` function of the C Standard Library, but it can be sent to the process from outside like any other signal.
SIGIOT indicates that the CPU has executed an explicit "trap" instruction (without a defined function), or an unimplemented instruction (when emulation is unavailable).

> *Note: "input/output trap" is a misnomer for any CPU "trap" instruction. The term reflects early usage of such instructions, predominantly to implement I/O functions, but they are not inherently tied to device I/O and may be used for other purposes such as communication between virtual & real hosts.*

SIGIOT and SIGABRT are typically the same signal, and receipt of that signal may indicate any of the conditions above.

`SIGALRM`, `SIGVTALRM` and `SIGPROF`

The SIGALRM, SIGVTALRM and SIGPROF signals are sent to a process when the corresponding time limit is reached. The process sets these time limits by calling `alarm` or `setitimer`. The time limit for SIGALRM is based on real or clock time; SIGVTALRM is based on CPU time used by the process; and SIGPROF is based on CPU time used by the process and by the system on its behalf (known as a *profiling timer*). On some systems SIGALRM may be used internally by the implementation of the `sleep` function.

`SIGBUS`

The SIGBUS signal is sent to a process when it causes a **bus error**. The conditions that lead to the signal being sent are, for example, incorrect memory access alignment or non-existent physical address.

`SIGCHLD`

The SIGCHLD signal is sent to a process when a **child process** terminates, is stopped, or resumes after being stopped. One common usage of the signal is to instruct the operating system to clean up the resources used by a child process after its termination without an explicit call to the `wait` system call.

`SIGCONT`

The SIGCONT signal instructs the operating system to **continue** (restart) a process previously paused by the SIGSTOP or SIGTSTP signal. One important use of this signal is in job control in the Unix shell.

`SIGFPE`

The SIGFPE signal is sent to a process when an exceptional (but not necessarily erroneous) condition has been detected in the floating-point or integer arithmetic hardware. This may include division by zero, floating-point underflow or overflow, integer overflow, an invalid operation or an inexact computation. Behaviour may differ depending on hardware.

`SIGHUP`

The SIGHUP signal is sent to a process when its controlling terminal is closed. It was originally designed to notify the process of a serial line drop (a **hangup**). In modern systems, this signal usually means that the controlling pseudo or virtual terminal has been closed.[11] Many daemons (who have no controlling terminal) interpret receipt of this signal as a request to reload their configuration files and flush/reopen their logfiles instead of exiting.[12] nohup is a command to make a command ignore the signal.

`SIGILL`

The SIGILL signal is sent to a process when it attempts to execute an **illegal**, malformed, unknown, or privileged instruction.

`SIGINT`

The SIGINT signal is sent to a process by its controlling terminal when a user wishes to **interrupt** the process. This is typically initiated by pressing `Ctrl+C`, but on some systems, the "delete" character or "break" key can be used.[13]

`SIGKILL`

The SIGKILL signal is sent to a process to cause it to terminate immediately (**kill**). In contrast to SIGTERM and SIGINT, this signal cannot be caught or ignored, and the receiving process cannot perform any clean-up upon receiving this signal. The following exceptions apply:

- Zombie processes cannot be killed since they are already dead and waiting for their parent processes to reap them.
- Processes that are in the blocked state will not die until they wake up again.
- The *init* process is special: It does not get signals that it does not want to handle, and thus it can ignore SIGKILL.[14] An exception from this rule is while init is ptraced on Linux.[15][16]
- An uninterruptibly sleeping process may not terminate (and free its resources) even when sent SIGKILL. This is one of the few cases in which a UNIX system may have to be rebooted to solve a temporary software problem.

SIGKILL is used as a last resort when terminating processes in most system shutdown procedures if it does not voluntarily exit in response to SIGTERM. To speed the computer shutdown procedure, Mac OS X 10.6, aka Snow Leopard, will send SIGKILL to applications that have marked themselves "clean" resulting in faster shutdown times with, presumably, no ill effects.[17] The command `killall -9` has a similar, while dangerous effect, when executed e.g. in Linux; it does not let programs save unsaved data. It has other options, and with none, uses the safer SIGTERM signal.

`SIGPIPE`

The SIGPIPE signal is sent to a process when it attempts to write to a pipe without a process connected to the other end.

`SIGPOLL`

The SIGPOLL signal is sent when an event occurred on an explicitly watched file descriptor.[18] Using it effectively leads to making asynchronous I/O requests since the kernel will **poll** the descriptor in place of the caller. It provides an alternative to active polling.

`SIGRTMIN` to `SIGRTMAX`

The SIGRTMIN to SIGRTMAX signals are intended to be used for user-defined purposes. They are **real-time** signals.

`SIGQUIT`

The SIGQUIT signal is sent to a process by its controlling terminal when the user requests that the process **quit** and perform a core dump.

`SIGSEGV`

The SIGSEGV signal is sent to a process when it makes an invalid virtual memory reference, or segmentation fault, i.e. when it performs a **seg**mentation **v**iolation.[19]

`SIGSTOP`

The SIGSTOP signal instructs the operating system to **stop** a process for later resumption.

`SIGSYS`

The SIGSYS signal is sent to a process when it passes a bad argument to a system call. In practice, this kind of signal is rarely encountered since applications rely on libraries (e.g. libc) to make the call for them. SIGSYS can be received by applications violating the

Linux Seccomp security rules configured to restrict them. SIGSYS can also be used to emulate foreign system calls, e.g. emulate Windows system calls on Linux.[20]

`SIGTERM`

The SIGTERM signal is sent to a process to request its **termination**. Unlike the SIGKILL signal, it can be caught and interpreted or ignored by the process. This allows the process to perform nice termination releasing resources and saving state if appropriate. SIGINT is nearly identical to SIGTERM.

`SIGTSTP`

The SIGTSTP signal is sent to a process by its controlling **terminal** to request it to **stop** (**t**erminal **st**o**p**). It is commonly initiated by the user pressing `Ctrl+Z`. Unlike SIGSTOP, the process can register a signal handler for, or ignore, the signal.

`SIGTTIN` and `SIGTTOU`

The SIGTTIN and SIGTTOU signals are sent to a process when it attempts to read **in** or write **out** respectively from the **tty** while in the background. Typically, these signals are received only by processes under job control; daemons do not have controlling terminals and, therefore, should never receive these signals.

`SIGTRAP`

The SIGTRAP signal is sent to a process when an exception (or **trap**) occurs: a condition that a debugger has requested to be informed of – for example, when a particular function is executed, or when a particular variable changes value.

`SIGURG`

The SIGURG signal is sent to a process when a socket has **urgent** or out-of-band data available to read.

`SIGUSR1` and `SIGUSR2`

The SIGUSR1 and SIGUSR2 signals are sent to a process to indicate **user-defined conditions**.

`SIGXCPU`

The SIGXCPU signal is sent to a process when it has used up the **CPU** for a duration that **exceeds** a certain predetermined user-settable value.[21] The arrival of a SIGXCPU signal provides the receiving process a chance to quickly save any intermediate results and to exit gracefully, before it is terminated by the operating system using the SIGKILL signal.

`SIGXFSZ`

The SIGXFSZ signal is sent to a process when it grows a **file** that **exceeds** the maximum allowed **size**.

`SIGWINCH`

The SIGWINCH signal is sent to a process when its controlling terminal changes its size (a **win**dow **ch**ange).[22]

Miscellaneous signals

The following signals are not specified in the POSIX specification. They are, however, sometimes used on various systems.

`SIGEMT`

The SIGEMT signal is sent to a process when an **emulator trap** occurs.

`SIGINFO`

The SIGINFO signal is sent to a process when a status (**info**) request is received from the controlling terminal.

`SIGPWR`

The SIGPWR signal is sent to a process when the system experiences a **power failure**.

SIGLOST

The SIGLOST signal is sent to a process when a file lock is **lost**.

SIGSTKFLT

The SIGSTKFLT signal is sent to a process when the coprocessor experiences a **st**ack **f**au**lt** (i.e. popping when the stack is empty or pushing when it is full).[23] It is defined by, but not used on Linux, where a x87 coprocessor stack fault will generate SIGFPE instead.[24]

SIGUNUSED

The SIGUNUSED signal is sent to a process when a system call with an **unused** system call number is made. It is synonymous with SIGSYS on most architectures.[23]

SIGCLD

The SIGCLD signal is synonymous with SIGCHLD.[23]

See also

- Asynchronous procedure call (APC)
- C signal handling

References

1. McIlroy, M. D. (1987). *A Research Unix reader: annotated excerpts from the Programmer's Manual, 1971–1986* (https://www.cs.dartmouth.edu/~doug/reader.pdf) (PDF) (Technical report). CSTR. Bell Labs. 139.
2. Gagliardi, Pietro. "C Programming in Plan 9 from Bell Labs" (https://doc.cat-v.org/plan_9/programming/c_programming_in_plan_9). *doc.cat-v.org*. Retrieved 22 January 2022.
3. "Termination Signals" (https://www.gnu.org/software/libc/manual/html_node/Termination-Signals.html). *The GNU C Library)*.
4. "Job Control Signals" (https://www.gnu.org/software/libc/manual/html_node/Job-Control-Signals.html). *The GNU C Library*.
5. "Miscellaneous Signals" (https://www.gnu.org/software/libc/manual/html_node/Miscellaneous-Signals.html). *The GNU C Library*.
6. "The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition: System Interfaces Chapter 2" (https://pubs.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_04.html#tag_02_04). *pubs.opengroup.org*. Retrieved 20 December 2020.
7. "signal(7) - Linux manual page" (https://man7.org/linux/man-pages/man7/signal.7.html). *man7.org*. Retrieved 20 December 2020.
8. "signal-safety(7) - Linux manual page" (https://man7.org/linux/man-pages/man7/signal-safety.7.html). *man7.org*. Retrieved 20 December 2020.
9. "The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition: <signal.h>" (https://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html). *pubs.opengroup.org*. Retrieved 20 December 2020.
10. "IEEE Std 1003.1-2017 - kill" (https://pubs.opengroup.org/onlinepubs/9699919799/utilities/kill.html). IEEE, Open Group. "The correspondence between integer values and the *sig* value used is shown in the following list. The effects of specifying any *signal_number* other than those listed below are undefined."
11. Michael Kerrisk (25 July 2009). "signal(7)" (https://www.kernel.org/doc/man-pages/online/pages/man7/signal.7.html). *Linux Programmer's Manual (version 3.22)*. The Linux Kernel Archives. Retrieved 23 September 2009.

12. "perlipc(1)" (https://perldoc.perl.org/perlipc.html#Handling-the-SIGHUP-Signal-in-Daemons). *Perl Programmers Reference Guide, version 5.18*. perldoc.perl.org - Official documentation for the Perl programming language. Retrieved 21 September 2013.
13. "Proper handling of SIGINT and SIGQUIT" (https://www.cons.org/cracauer/sigint.html). Retrieved 6 October 2012.
14. https://manpages.ubuntu.com/manpages/zesty/man2/kill.2.html Archived (https://web.archive.org/web/20180128103316/https://manpages.ubuntu.com/manpages/zesty/man2/kill.2.html) 28 January 2018 at the Wayback Machine section NOTES
15. "SIGKILL init process (PID 1)" (https://stackoverflow.com/a/21031583). *Stack Overflow*.
16. "Can root kill init process?" (https://unix.stackexchange.com/a/308429). *Unix & Linux Stack Exchange*.
17. "Mac Dev Center: What's New in Mac OS X: Mac OS X v10.6" (https://developer.apple.com/mac/library/releasenotes/MacOSX/WhatsNewInOSX/Articles/MacOSX10_6.html#//apple_ref/doc/uid/TP40008898-SW22). 28 August 2009. Retrieved 18 November 2017.
18. "ioctl - controls a STREAM device" (https://pubs.opengroup.org/onlinepubs/9699919799/functions/ioctl.html). *POSIX system call specification*. The Open Group. Retrieved 19 June 2015.
19. "What is a "segmentation violation"?" (https://support.microfocus.com/kb/doc.php?id=7001662). *support.microfocus.com*. Retrieved 22 November 2018.
20. "Syscall User Dispatch – The Linux Kernel documentation" (https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html). *kernel.org*. Retrieved 11 February 2021.
21. "getrlimit, setrlimit - control maximum resource consumption" (https://pubs.opengroup.org/onlinepubs/009695399/functions/getrlimit.html). *POSIX system call specification*. The Open Group. Retrieved 10 September 2009.
22. Clausecker, Robert (19 June 2017). "0001151: Introduce new signal SIGWINCH and functions tcsetsize(), tcgetsize() to get/set terminal window size" (http://austingroupbugs.net/view.php?id=1151). *Austin Group Defect Tracker*. Austin Group. Retrieved 12 October 2017. "Accepted As Marked"
23. "signal(7) — Linux manual pages" (https://manpages.courier-mta.org/htmlman7/signal.7.html). *manpages.courier-mta.org*. Retrieved 22 November 2018.
24. "Linux 3.0 x86_64: When is SIGSTKFLT raised?" (https://stackoverflow.com/questions/9332864/linux-3-0-x86-64-when-is-sigstkflt-raised#comment51557777_9333099). *Stack Overflow*.

- Stevens, W. Richard (1992). *Advanced Programming in the UNIX® Environment* (https://archive.org/details/advancedprogramm00stev). Reading, Massachusetts: Addison Wesley. ISBN 0-201-56317-7.
- "The ®Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition" (https://pubs.opengroup.org/onlinepubs/9799919799/). The Open Group. Retrieved 2 March 2025.

External links

- Unix Signals Table, Ali Alanjawi, University of Pittsburgh (https://people.cs.pitt.edu/~alanjawi/cs449/code/shell/UnixSignals.htm)
- Man7.org Signal Man Page (http://man7.org/linux/man-pages/man7/signal.7.html)
- Introduction To Unix Signals Programming Introduction To Unix Signals Programming (https://web.archive.org/web/20130926005901/http://users.actcom.co.il/~choo/lupg/tutorials/signals/signals-programming.html) at the Wayback Machine (archived 26 September 2013)
- Another Introduction to Unix Signals Programming (https://www.linuxprogrammingblog.com/all-about-linux-signals) (blog post, 2009)

- UNIX and Reliable POSIX Signals (https://web.archive.org/web/20230413214743/http://www.enderunix.org/docs/signals.pdf)[usurped] by Baris Simsek (stored by the Internet Archive)
- Signal Handlers (https://www.openbsd.org/papers/opencon04/index.html) by Henning Brauer

---