WIKIPEDIA

Signal (IPC)

Signals 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 in order to notify it of an event that occurred. Signals originated in 1970s Bell Labs Unix and have been more recently specified in the POSIX standard.

When a signal is sent, the operating system interrupts the target process' 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 interprocess communications, as the computational and memory footprint for signals is small.

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

Contents

History

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

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 or a segmentation violation will generate signals (here, SIGFPE "floating point exception" and SIGSEGV "segmentation violation" respectively, which both by default cause a core dump and a program exit).

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:

- 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.
- 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.

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 (http://pubs.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_04.html#tag_02_04) specification and the Linux man page `signal(7)` (http://man7.org/linux/man-pages/man7/signal.7.html) requires that all system functions directly or *indirectly* called from a signal function are *async-signal safe* and gives a list of such async-signal safe system functions (practically the system calls), otherwise it is an undefined behavior. It is suggested (http://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html) to simply set some `volatile sig_atomic_t` variable in a signal handler, and to test it elsewhere.

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 TLB miss.

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 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. All signals are defined as macro constants in `<signal.h>` header file. The name of the macro constant consists of a "SIG" prefix followed by a mnemonic name for the signal.

`SIGABRT` and `SIGIOT`
: The SIGABRT and SIGIOT 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.

`SIGALRM`, `SIGVTALRM` and `SIGPROF`
: The SIGALRM, SIGVTALRM and SIGPROF signal is sent to a process when the time limit specified in a call to a preceding **alarm** setting function (such as `setitimer`) elapses. SIGALRM is sent when real or clock time elapses. SIGVTALRM is sent when CPU time used by the process elapses. SIGPROF is sent when CPU time used by the process and by the system on behalf of the process elapses.

`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 interrupted, or resumes after being interrupted. 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 it executes an erroneous arithmetic operation, such as division by zero (the name "FPE", standing for floating-point exception, is a misnomer as the signal covers integer-arithmetic errors as well).[2]

`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.[3] Many daemons will reload their configuration files and reopen their logfiles instead of exiting when receiving this signal.[4] 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.[5]

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.[6] An exception from this exception is while init is ptraced on Linux.[7][8]
- 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.[9]

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.[10] 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**.[11]

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.

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.[12] 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** than 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).[13]

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

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 (not terminate) the process.

Continue — Continue the process, if it is stopped; otherwise, ignore the signal.

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**ac**k f**au**lt** (i.e. popping when the stack is empty or pushing when it is full).[15] It is defined by, but not used on Linux, where a x87 coprocessor stack fault will generate SIGFPE instead.[16]

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.[15]

SIGCLD
The SIGCLD signal is synonymous with SIGCHLD.[15]

See also

- C signal handling

References

1. McIlroy, M. D. (1987). *A Research Unix reader: annotated excerpts from the Programmer's Manual, 1971–1986* (http://www.cs.dartmouth.edu/~doug/reader.pdf) (PDF) (Technical report). CSTR. Bell Labs. 139.
2. Free Software Foundation, Inc. (2014-02-09). "Program Error Signals" (https://www.gnu.org/software/libc/manual/html_node/Program-Error-Signals.html). *The GNU C Library Reference Manual (version 2.19)*. Free Software Foundation. Retrieved 2014-03-01.
3. 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.
4. "perlipc(1)" (http://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.
5. "Proper handling of SIGINT and SIGQUIT" (http://www.cons.org/cracauer/sigint.html). Retrieved 6 October 2012.
6. http://manpages.ubuntu.com/manpages/zesty/man2/kill.2.html section NOTES
7. "SIGKILL init process (PID 1)" (https://stackoverflow.com/a/21031583). *Stack Overflow*.
8. "Can root kill init process?" (https://unix.stackexchange.com/a/308429). *Unix & Linux Stack Exchange*.
9. "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). 2009-08-28. Retrieved 18 November 2017.
10. "ioctl - controls a STREAM device" (http://pubs.opengroup.org/onlinepubs/9699919799/functions/ioctl.html). *POSIX system call specification*. The Open Group. Retrieved 19 June 2015.
11. "What is a "segmentation violation"? Novell Support" (http://www.novell.com/support/kb/doc.php?id=7001662). Retrieved 8 February 2013.

12. "getrlimit, setrlimit - control maximum resource consumption" (http://www.opengroup.org/onlinepubs/009695399/functions/getrlimit.html). *POSIX system call specification*. The Open Group. Retrieved 10 September 2009.
13. Clausecker, Robert (2017-06-19). "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 2017-10-12. "Accepted As Marked"
14. "IEEE Std 1003.1 - kill" (http://pubs.opengroup.org/onlinepubs/009695399/utilities/kill.html). IEEE, Open Group. "The correspondence between integer values and the sig value used is shown in the following table. The effects of specifying any signal_number other than those listed in the table are undefined."
15. "signal(7) — Linux manual pages" (http://manpages.courier-mta.org/htmlman7/signal.7.html). *manpages.courier-mta.org*.
16. "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* (http://www.kohala.com/start/apue.html). Reading, Massachusetts: Addison Wesley. ISBN 0-201-56317-7.
- "The Open Group Base Specifications Issue 7, 2013 Edition" (http://pubs.opengroup.org/onlinepubs/9699919799/). The Open Group. Retrieved 19 June 2015.

External links

- Unix Signals Table, Ali Alanjawi, University of Pittsburgh (http://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 (http://www.linuxprogrammingblog.com/all-about-linux-signals)
- UNIX and Reliable POSIX Signals (http://www.enderunix.org/docs/signals.pdf) by Baris Simsek
- Signal Handlers (http://www.openbsd.org/papers/opencon04/index.html) by Henning Brauer

---