Unix Signals

News	See Also	Books	Recommended Links	Controlling System Processes in Solaris	kill command
SIGKILL	SIGTERM	Zombies	process table		
		Admin Horror Stories	Unix History	Humor	Etc

Signals are a simple Unix mechanism for controlling processes. A *signal* is a 6-bit message to a process that requires immediate attention. Each signal has a default action associated with it; for some signals, you can change this default action. Signals are generated by exceptions, which include:

- Attempts to use illegal instructions
- Certain kinds of mathematical operations
- Window resize events
- Predefined alarms, including expiration of a timer
- The user pressing an interrupt key on a terminal
- Another program using the kill( ) or killpg( ) system calls
- A program running in the background attempting to read from or

write to its controlling terminal

- A child process calling exit or terminating abnormally

A complete list of signals that the kill command can send can be found by executing the command kill -l, or by referring to the man page for signal. For example on Solaris:

```
# man -s3head signal
```

You can also obtain the list of the signals for particular flavour of Unix from iether man signal or by viewing sygnal.h system header. On Solaris the latter is located at /usr/include/signal.h

The system default may be to ignore the signal, to terminate the process receiving the signal (and, optionally, generate a core file), or to suspend the process until it receives a continuation signal. Some signals can be *caught*—that is, a program can specify a particular function that should be run when the signal is received. As originally designed, Unix supports exactly 31 signals. Most modern flavours of Unix have extended this set to include more signals, typically 63.

The signals and types are usually listed in the files */usr/include/signal.h* and */usr/include/sys/signal.h*. For example in Solaris we have (A Primer on Signals in the Solaris OS):

Name	Number	Default action	Description
SIGHUP	1	**Exit**	**Hangup** (ref termio(7I)).Usually means that the controlling terminal has been disconnected.
SIGINT	2	**Exit**	Interrupt (ref termio(7I)).The user can generate this signal by pressing Ctrl+C or Delete.
SIGQUIT	3	**Core**	Quit (ref termio(7I)). Quits the process and produces a core dump.
SIGILL	4	**Core**	Illegal Instruction
SIGTRAP	5	**Core**	Trace or breakpoint trap
SIGABRT	6	**Core**	Abort
SIGEMT	7	**Core**	Emulation trap
SIGFPE	8	**Core**	Arithmetic exception. Informs a process of a floating-point error.
SIGKILL	9	**Exit**	**Kill.** Forces the process to terminate. This is a sure kill.
SIGBUS	10	**Core**	Bus error -- actually a misaligned address error
SIGSEGV	11	**Core**	Segmentation fault -- an address reference boundary error

SIGSYS	12	**Core**	Bad system call
SIGPIPE	13	**Exit**	Broken pipe
SIGALRM	14	**Exit**	Alarm clock
SIGTERM	15	**Exit**	**Terminated.** A gentle kill that gives processes a chance to clean up
SIGUSR1	16	**Exit**	User defined signal 1
SIGUSR2	17	**Exit**	User defined signal 2
SIGCHLD	18	**Ignore**	**Child process status changed**
SIGPWR	19	**Ignore**	Power fail or restart
SIGWINCH	20	**Ignore**	Window size change
SIGURG	21	**Ignore**	Urgent socket condition
SIGPOLL	22	**Exit**	Pollable event (ref streamio(7I))
SIGSTOP	23	**Stop**	Stop (cannot be caught or ignored)
SIGTSTP	24	**Stop**	Stop (job control, e.g., ^z))
SIGCONT	25	**Ignore**	Continued
SIGTTIN	26	**Stop**	Stopped -- tty input (ref termio(7I))
SIGTTOU	27	**Stop**	Stopped -- tty output (ref termio(7I))
SIGVTALRM	28	**Exit**	Virtual timer expired
SIGPROF	29	**Exit**	Profiling timer expired
SIGXCPU	30	**Core**	CPU time limit exceeded (ref getrlimit(2))
SIGXFSZ	31	**Core**	File size limit exceeded (ref getrlimit(2))
SIGWAITING	32	**Ignore**	Concurrency signal used by threads library
SIGLWP	33	**Ignore**	Inter-LWP signal used by threads library
SIGFREEZE	34	**Ignore**	Checkpoint suspend
SIGTHAW	35	**Ignore**	Checkpoint resume
SIGCANCEL	36	**Ignore**	Cancellation signal used by threads library
SIGLOST	37	**Ignore**	Resource lost

SIGRTMIN	38	**Exit**	Highest priority realtime signal
SIGRTMAX	45	**Exit**	Lowest priority realtime signal

The symbols in the "Key" column of table above have the following meaning:

- * If signal is not caught or ignored, generates a *core image* dump
- @ Signal is ignored by default
- + Signal causes process to suspend
- ! Signal cannot be caught or ignored

Signals are normally used between processes for process control. They are also used within a process to indicate exceptional conditions that should be handled immediately (for example, floating-point overflows).

Unix Signals and the kill Command

The Unix superuser can use the kill command to terminate any process on the system. One of the most common uses of the kill command is to kill a "runaway" process that is consuming CPU and memory for no apparent reason. You may also want to kill the processes belonging to an intruder.

Despite its name, the kill command can be used for more than simply terminating processes. The kill command can send any signal to any process. Although some signals do indeed result in processes being terminated, others can cause a process to stop, restart, or perform other functions.

The syntax of the kill command is:

kill [*-signal*] *process-IDs*

The kill command allows signals to be specified by number or name. To send a hangup to process #1, for example, type:

 # kill -HUP 1

With some older versions of Unix, signals could be specified only by number; all versions of the kill command still accept this syntax as well:

 # kill -1 1

The superuser can kill any process; other users can kill only their own processes. You send sygnalk to several processes by listing all of their PIDs on the command line:

 # kill -HUP 1023 3421 3221

By default, kill sends SIGTERM (signal 15), the process-terminate signal.

Killing Multiple Processes at the Same Time

Modern Unix systems allow you to send a signal to multiple processes at the same time with the kill command:

- If you specify 0 as the PID, the signal is sent to all the processes in your process group.
- If you specify -1 as a PID and you are not the superuser, the signal is sent to all processes having the same UID as you.
- If you specify -1 as a PID and you are the superuser, the signal is sent to all processes except system processes, process #1, and yourself.
- If you specify any other negative value, the signal is sent to all processes in the process group numbered the same as the absolute value of your argument.

Catching Signals

Many signals, including SIGTERM, can be caught by programs. When catching a signal, a programmer has three choices of what to do with the signal:

- Ignore it.
- Perform the default action.
- Execute a program-specified function, often called a *signal handler*.

Signal handling gives Unix programs a lot of flexibility. For example, some programs catch SIGINT (signal 2), sent when the user types Ctrl-C, to save their temporary files before exiting; other programs perform the default action and simply exit.

There are two signals that cannot be caught: SIGKILL (signal 9) and SIGSTOP (signal 17). SIGKILL terminates a program, no questions asked. SIGSTOP causes a program to stop execution dead in its tracks.

One signal that is very often sent is SIGHUP (signal 1), which simulates a hangup on a modem. Because having a modem accidentally hung up was once a common occurrence, many programs catch SIGHUP and perform a clean shutdown. Standard practice when killing a process is to send signal 1 (hangup) first; if the process does not terminate, then send it signal 15 (software terminate), and finally signal 9 (sure kill).

Many system programs catch SIGHUP and use it as a signal to re-read their configuration files. This has become a common programming

convention, particularly in programs that don't expect to interact with a modem, such as network daemons.

Killing Rogue or Questionable Processes

Sometimes simply killing a rogue process is the wrong thing to do: you can learn more about a process by stopping it and examining it with some of Unix's debugging tools than by "blowing it out of the water." Sending a process a SIGSTOP will stop the process but will not destroy the process's memory image. This will allow you to examine the process.

Old News ;-)

Hour 19 Dealing with Signals How Are Signal Represented

Dealing with Signals

Signals are software interrupts sent to a program to indicate that an important event has occurred. The events can vary from user requests to illegal

memory access errors. Some **signals**, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control.

Because **signals** can arrive at any time during the execution of a script, they add an extra level of complexity to shell scripts. Scripts must account for this fact and include extra code that can determine how to respond appropriately to a signal regardless of what the script was doing when the signal was received.

In this chapter you will look at the following topics:

- The different types of **signals** encountered in shell programming
- How to deliver **signals** using the kill command
- Handling **signals**
- How to use **signals** within your script

How Are Signal Represented?

Getting a List of **Signals** Delivering **Signals**

Default Actions

Each type of event is represented by a separate signal. Each signal is only a small positive integer. The **signals** most commonly encountered in shell script programming are given in Table 19.1. All the listed **signals** are available on all versions of UNIX.

Table 19.1 Important Signals for Shell Scripts

Name	Value	Description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Interrupt from keyboard
SIGQUIT	3	Quit from keyboard
SIGKILL	9	Kill signal
SIGALRM	14	Alarm Clock signal (used for timers)
SIGTERM	15	Termination signal

In addition to the **signals** listed in Table 19.1, you might occasionally see a reference to signal 0, which is more of a shell convention than a real signal. When a shell script exits either by using the exit command or by executing the last command in the script, the shell in which the script was running sends itself a signal 0 to indicate that it should terminate.

Getting a List of Signals

All the **signals** understood by your system are listed in the C language header file signal.h. The location of this file varies between UNIX flavors. Some common locations are

- **Solaris** and HPUX: /usr/include/sys/signal.h
- Linux: /usr/include/asm/signal.h

Some vendors provide a man page for this file which you can view with one of the following commands:

- In Linux: man 7 signal
- In **Solaris**: man -s 5 signal
- In HP-UX: man 5 signal

Another way that your system can understand a list of **signals** is to use the -l option of the kill command. For example on a **Solaris** system the output is:

```
$ kill -l
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
 9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGUSR1
17) SIGUSR2     18) SIGCHLD     19) SIGPWR      20) SIGWINCH
21) SIGURG      22) SIGIO       23) SIGSTOP     24) SIGTSTP
25) SIGCONT     26) SIGTTIN     27) SIGTTOU     28) SIGVTALRM
29) SIGPROF     30) SIGXCPU     31) SIGXFSZ     32) SIGWAITING
33) SIGLWP      34) SIGFREEZE   35) SIGTHAW     36) SIGCANCEL
37) SIGLOST
```

The actual list of **signals** varies between **Solaris**, HP-UX, and Linux.

Default Actions

Every signal, including those listed in Table 19.1, has a *default action* associated with it. The default action for a signal is the action that a script or program performs when it receives a signal.

Some of the possible default actions are

- Terminate the process.
- Ignore the signal.
- Dump core. This creates a file called core containing the memory image of the process when it received the signal.
- Stop the process.
- Continue a stopped process.

The default action for the **signals** that you should be concerned about is to terminate the process. Later in this chapter you will look at how you can change the default action performed by a script with a *signal handler* .

Delivering Signals

There are several methods of delivering **signals** to a program or script. One of the most common is for a user to type CONTROL-C or the INTERRUPT key while a script is executing. In this case a SIGINT is sent to the script and it terminates.

The other common method for delivering **signals** is to use the kill command as follows:

```
kill -signal pid
```

Here *signal* is either the number or name of the signal to deliver and *pid* is the process ID that the signal should be sent to.

TERM

In previous chapters you looked at the kill command without the signal argument. By default the kill command sends a TERM or terminates a signal to the program running with the specified pid. Recall from Chapter 6, "Processes," that a PID is the process ID given by UNIX to a program while it is executing. Thus the commands

```
kill pid
kill -s SIGTERM pid
```

are equivalent.

Now look at a few examples of using the kill command to deliver other **signals**.

HUP

The following command

```
$ kill -s SIGHUP 1001
```

sends the HUP or hang-up signal to the program that is running with process ID 1001. You can also use the numeric value of the signal as follows:

```
$ kill -1 1001
```

This command also sends the hang-up signal to the program that is running with process ID 1001. Although the default action for this signal calls for the process to terminate, many UNIX programs use the HUP signal as an indication that they should reinitialize themselves. For this reason, you should use a different signal if you are trying to terminate or kill a process.

QUIT and INT

If the default kill command cannot terminate a process, you can try to send the process either a QUIT or an INT (interrupt) signal as follows:

```
$ kill -s SIGQUIT 1001
```

or

```
$ kill -s SIGINT 1001
```

One of these **signals** should terminate a process, either by asking it to quit (the QUIT signal) or by asking it to interrupt its processing (the INT signal).

kill

Some programs and shell scripts have special functions called *signal handlers* that can ignore or discard these **signals**. To terminate such a program, use the kill signal:

```
$ kill -9 1001
```

Here you are sending the kill signal to the program running with process ID 1001. In this case you are using the numeric value of the signal, instead of the name of the signal. By convention, the numeric value of the kill signal, 9, is always used for it.

The kill signal has the special property that it cannot be caught, thus any process receiving this signal terminates immediately "with extreme prejudice." This means that a process cannot cleanly exit and might leave data it was using in a corrupted state. You should only use this signal to terminate a process when all the other **signals** fail to do so.

[Mar 03, 2011] A Primer on Signals in the Solaris OS by Jim Mauro

2001 | Sun/Oracle

Signals are used to notify a process or thread of a particular event. Many engineers compare signals with hardware interrupts, which occur when a hardware subsystem such as a disk I/O interface (an SCSI host adapter, for example) generates an interrupt to a processor as a result of a completed I/O. This event in turn causes the processor to enter an interrupt handler, so subsequent processing can be done in the operating system based on the source and cause of the interrupt.

UNIX guru W. Richard Stevens, however, aptly describes signals as software interrupts . When a signal is sent to a process or thread, a signal handler may be entered (depending on the current disposition of the signal), which is similar to the system entering an interrupt handler as the result of receiving an interrupt.

There is quite a bit of history related to signals, design changes in the signal code, and various implementations of UNIX. This was due in part to some deficiencies in the early implementation of signals, as well as the parallel development work done on different versions of UNIX, primarily BSD UNIX and AT&T System V. W. Richard Stevens, James Cox, and Berny Goodheart (see Resources) cover these details in their respective books. What *does* warrant mention is that early implementations of signals were deemed unreliable. The unreliability stemmed from the fact that in the old days the kernel would reset the signal handler to its default if a process caught a signal and invoked its own handler, and the reset occurred before the handler was invoked. Attempts to address this issue in user code by having the signal handler first reinstall itself did not always solve the problem, as successive occurrences of the same signal resulted in race conditions, where the default action was invoked before the user-defined handler was reinstalled. For signals that had a default action of terminating the process, this created severe problems. This problem (and some others) were addressed in 4.3BSD UNIX and SVR3 in the mid-'80s.

The implementation of reliable signals has been in place for many years now, where an installed signal handler remains persistent and is not reset by the kernel. The POSIX standards provided a fairly well-defined set of interfaces for using signals in code, and today the Solaris Operating Environment implementation of signals is fully POSIX-compliant. Note that reliable signals require the use of the newer sigaction(2) interface, as opposed to the traditional signal(3C) call.

The occurrence of a signal may be synchronous or asynchronous to the process or thread, depending on the source of the signal and the underlying reason or cause. Synchronous signals occur as a direct result of the executing instruction stream, where an unrecoverable error (such as an illegal instruction or illegal address reference) requires an immediate termination of the process. Such signals are directed to the thread whose execution stream caused the error. Because an error of this type causes a trap into a kernel trap handler, synchronous signals are sometimes referred to as *traps.* Asynchronous signals are external to (and in some cases unrelated to) the current execution context. One obvious example is the sending of a signal to a process from another process or thread, via a kill(2), _lwp_kill(2), or sigsend(2) system call, or a thr_kill(3T), pthread_kill(3T), or sigqueue(3R) library invocation. Asynchronous signals are also referred to as interrupts.

Every signal has a unique signal name, an abbreviation that begins with SIG (SIGINT for interrupt signal, for example) and a corresponding signal number. Additionally, for all possible signals, the system defines a default disposition, or action to take when a signal occurs. There are four possible default dispositions:

- **Exit**: Forces the process to exit
- **Core**: Forces the process to exit, and creates a core file
- **Stop**: Stops the process
- **Ignore**: Ignores the signal; no action taken

A signal's disposition within a process's context defines what action the system will take on behalf of the process when a signal is delivered. All threads and LWPs (lightweight processes) within a process share the signal disposition, which is processwide and cannot be unique among threads within the same process. The table below provides a complete list of signals, along with a description and default action.

The disposition of a signal can be changed from its default, and a process can arrange to catch a signal and invoke a signal handling routine of its own, or ignore a signal that may not have a default disposition of Ignore. The only exceptions are SIGKILL and SIGSTOP, whose default dispositions cannot be changed. The interfaces for defining and changing signal disposition are the signal(3C) and sigset(3C) libraries, and the sigaction(2) system call. Signals can also be blocked, which means the process has temporarily prevented delivery of a signal. The generation of a signal that has been blocked will result in the signal remaining pending to the process until it is explicitly unblocked, or the disposition is changed to Ignore. The sigprocmask(2) system call will set or get a process's signal mask, the bit array that is inspected by the kernel to determine if a signal is blocked or not. thr_setsigmask(3T) and pthread_sigmask(3T) are the equivalent interfaces for setting and retrieving the signal mask at the user-threads level.

I mentioned earlier that a signal may originate from several different places, for a variety of different reasons. The first three signals listed in the table above - SIGHUP, SIGINT, and SIGQUIT - are generated by a keyboard entry from the controlling terminal (SIGINT and SIGHUP), or they are generated if the control terminal becomes disconnected (SIGHUP - use of the nohup(1) command makes processes "immune" from hangups by setting the disposition of SIGHUP to Ignore). Other terminal I/O-related signals include SIGSTOP, SIGTTIN, SIGTTOU, and SIGTSTP. For the signals that originate from a keyboard command, the actual key sequence that generates the signals, usually Ctrl-C, is defined within the parameters of the terminal session, typically via stty(1), which results in a SIGINT being sent to a process, and has a default disposition of Exit.

Signals generated as a direct result of an error encountered during instruction execution start with a hardware trap on the system. Different processor

architectures define various traps that result in an immediate vectored transfer of control to a kernel trap-handling function. The Solaris kernel builds a trap table and inserts trap-handling routines in the appropriate locations based on the architecture specification of the processors that Solaris supports: SPARC V7 (early Sun-4 architectures), SPARC V8 (SuperSPARC - Sun-4m and Sun-4d architectures), SPARC V9 (UltraSPARC), and x86 (in Intel parlance they're called *interrupt descriptor tables* or IDTs; on SPARC, they're called *trap tables*). The kernel-installed trap handler will ultimately generate a signal to the thread that caused the trap. The signals that result from hardware traps are SIGILL, SIGFPE, SIGSEGV, SIGTRAP, SIGBUS, and SIGEMT.

In addition to terminal I/O and error trap conditions, signals can originate from sources such as an explicit send programmatically via kill(2) or thr_kill(3T), or from a shell issuing a kill(1) command. Parent processes are notified of status change in a child process via SIGCHLD. The alarm(2) system call sends a SIGALRM when the timer expires. Applications can create user-defined signals as a somewhat crude form of interprocess communication by defining handlers for SIGUSR1 or SIGUSR2 and then sending those signals between processes. The kernel sends SIGXCPU if a process exceeds its processor time resource limit or SIGXFSZ if a file write exceeds the file size resource limit. A SIGABRT is sent as a result of an invocation of the abort(3C) library. If a process is writing to a pipe and the reader has terminated, SIGPIPE is generated.

These examples of signals generated as a result of events beyond hard errors and terminal I/O do not represent the complete list, but rather provide you with a well-rounded set of examples of the process-induced and external events that can generate signals. You can find a complete list in any number of texts on UNIX programming.

In terms of actual implementation, a signal is represented as a bit in a data structure (several data structures, actually, as you'll see shortly). More succinctly, the posting of a signal by the kernel results in a bit getting set in a structure member at either the process or thread level. Because each signal has a unique signal number, a structure member of sufficient width is used, which allows every signal to be represented by simply setting the bit that corresponds to the signal number of the signal you wish to post (for example, setting the 17th bit to post signal 17, SIGUSR1).

Because Solaris includes more than 32 possible signals, a long or int data type is not sufficiently wide to represent each possible signal as a unique bit, so a data structure is required. The k_sigset_t data structure defined in /usr/include/signal.h is used in several of the process data structures to store the posted signal bits. It's an array of two unsigned long data types (array members 0 and 1), providing a bit width of 64 bits.

- InformIT Solaris 10 System Administration Exam Prep Managing System Processes Using Signals
- Signal (computing) - Wikipedia, the free encyclopedia

    - SIGKILL - Wikipedia, the free encyclopedia
    - SIGTERM - Wikipedia, the free encyclopedia
- Zombie process - Wikipedia, the free encyclopedia
- Commands
    - **kill command**
    - killall - on some variations of Unix, such as Solaris, this utility is automatically invoked when the system is going through a shutdown. It behaves much like the kill command above, but instead of sending a signal to an individual process, the signal is sent to all processes on the system. However, on others such as IRIX, Linux, and FreeBSD, an argument is supplied specifying the name of the process (or processes) to kill. For instance, to kill a process such as an instance of the XMMS music player invoked by `xmms`, the user would run the command `killall xmms`. This would kill all processes named `xmms`.
    - pkill - signals processes based on name and other attributes. It was introduced in Solaris 7 and has since been reimplemented for Linux and OpenBSD. pkill makes killing processes based on their name much more convenient: e.g. to kill a process named *firefox* without pkill (and without pgrep), one would have to type `kill `ps --no-headers -C firefox -o pid`` whereas with pkill, one can simply type `pkill firefox`.
    - xkill - if called without any parameters, the cursor changes, and the user can click on a window to kill the underlying process.
- **Books**
    - *Advanced Programming in the UNIX Environment* (Addison-Wesley, ISBN 0-201-56317-7) Stevens, W. Richard.
    - *Multithreaded Programming with Pthreads* (Sun Microsystems Press/Prentice Hall ISBN 0-13-680729-1) Berg, Daniel, J. and Lewis, Bill.
    - *Programming with Threads* (Sun Microsystems Press/Prentice Hall, ISBN 0-13-172389-8) Kleiman, Steve, Shah, Devang, and Smaalders, Bart.
    - *The Magic Garden Explained: The Internals of UNIX System V Release 4* (Prentice Hall, ISBN 0-13-098138-9) Goodheart, Berny, Cox, James, and Mashey, John, R.
    - *UNIX Internals: The New Frontiers* (Prentice Hall, ISBN 0-13-101908-2) Vahalia, Uresh.

Etc

The Last but not Least

whom English is not a native language. Grammar and spelling errors should be expected. **The site contain some broken links as it develops like a living tree...**

You can use Amazon HonorSystem to make a contribution, supporting this site

Disclaimer:

The statements, views and opinions presented on this web page are those of the author and are not endorsed by, nor do they necessarily reflect, the opinions of the author present and former employers, SDNP or any other organization the author may be associated with. We do not warrant the correctness of the information provided or its fitness for any purpose.

Last modified: January, 27, 2013