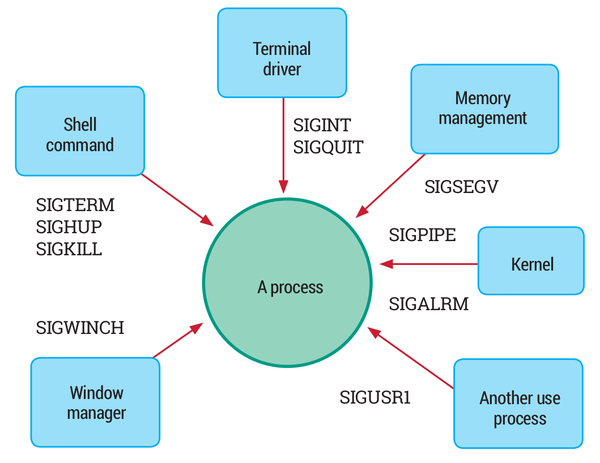
**Linux SIgnals**

SUMMARY

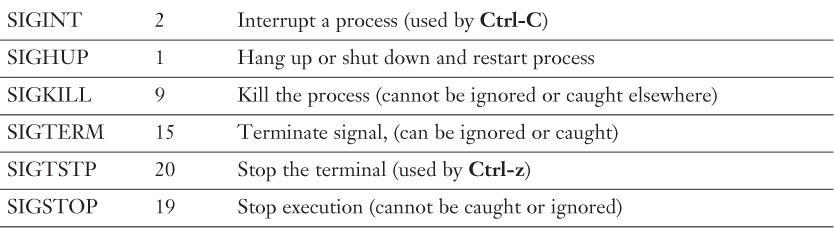
Common sources of Linux signals. Source: Brown 2015.

A Linux computer system has many processes in different states. These processes belong to either user applications or the operating system. We need a mechanism for the kernel and these processes to coordinate their activities. One way to do this is for a process to inform others when something important happens. This is why we have **signals**.

A signal is basically a one-way notification. A signal can be sent by the kernel to a process, by a process to another process, or a process to itself.

Linux signals trace their origins to Unix signals. In later Linux versions, real-time signals were added. Signals are a simple and lightweight form of inter-process communication, and therefore suited for embedded systems.

DISCUSSION

1. What are the essential facts about Linux signals?

Describing a few important Linux signals. Source: Walberg and Brunson 2015, table 7-2.

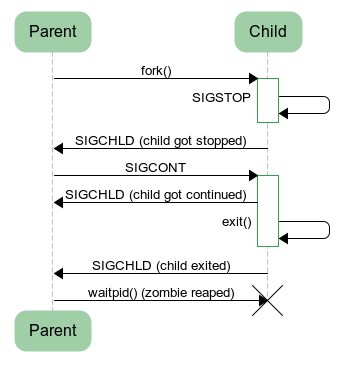
There are 31 standard signals, numbered 1-31. Each signal is named as "SIG" followed by a suffix. Starting from version 2.2, the Linux kernel supports 33 different real-time signals. These have numbers 32-64 but programmers should instead use SIGRTMIN+n notation. Standard signals have specific purposes but the use of SIGUSR1 and SIGUSR2 can be defined by applications. Real-time signals are also defined by applications.

Signal number 0, which POSIX.1 calls *null signal*, is generally not used but kill function uses this as a special case. No signal is sent but it can be used (rather unreliably) to check if the process still exists.

Linux implementation of signals is fully POSIX compliant. Newer implementation should prefer to use sigaction rather than the traditional signal interface.

Just as hardware subsystems can interrupt the processor, signals interrupt process execution. They are therefore seen as *software interrupts*. While interrupt handlers process hardware interrupts, signal handlers process signals.

Some signals are mapped to specific key inputs: SIGINT for ctrl+c, SIGSTOP for ctrl+z, SIGQUIT for ctrl+\.

1. How does a signal affect the state of a Linux process?

Example exchange of signals between parent and child processes. Source: Marek 2012.

Some signals terminate the receiving process: SIGHUP, SIGINT, SIGTERM, SIGKILL. There are signals that terminate the process along with a core dump to help programmers debug what went wrong: SIGABRT (abort signal), SIGBUS (bus error), SIGILL (illegal instruction), SIGSEGV (invalid memory reference), SIGSYS (bad system call). Other signals stop the process: SIGSTOP, SIGTSTP. SIGCONT is a signal that resumes a stopped process.

A program could override default behaviour. For example, an interactive program could be written to ignore SIGINT (generated by ctrl+c input). Two notable exceptions are SIGKILL and SIGSTOP signals, which can't be ignored, blocked or overridden this way.

Let's consider the example of a parent process and its child process. Suppose the child sends SIGSTOP to itself, child process will be stopped. This in turns triggers SIGCHLD to the parent. Parent can then signal the child to continue using SIGCONT. When child comes out of stopped state, another SIGCHLD is sent to parent. If later, the child process exits, the final SIGCHLD is sent to parent.

1. Aren't signals similar to exceptions?

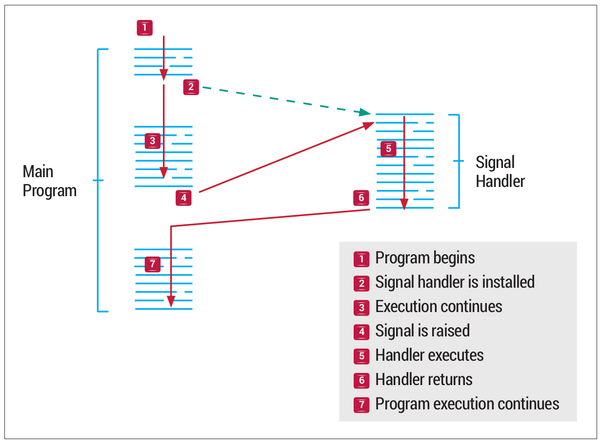
Some programming languages are capable of exceptions using constructs such as try-throw-catch. Signals are not similar to exceptions. Instead, failed system or library calls return non-zero exit codes. When a process is terminated, it's exit code will be 128 plus signal number. For example, a process killed by SIGKILL will return 137 (128+9).

1. Are Linux signals synchronous or asynchronous?

When signals are generated, they can be considered as synchronous or asynchronous.

**Synchronous** signals occur as a result of instructions that have led to an unrecoverable error such as an illegal address access. These signals are sent to the thread that caused it. These are also called *traps*, since they also cause a trap into the kernel trap handler.

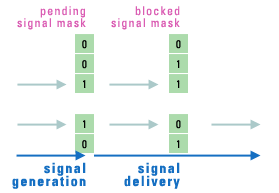
**Asynchronous** signals are external to the current execution context. Sending SIGKILL from an another process is an example of this. These are also called *software interrupts*.

1. What's the typical lifecycle of a signal?

Signals when handled interrupt program execution. Source: Brown 2015.

A signal goes through three stages:

* + **Generation**: A signal can be generated by the kernel or any of the processes. Whoever generates the signal, addresses it to a specific process. A signal is represented by its number and has no extra data or arguments. Thus, signals are lightweight. However, extra data can be passed along for POSIX real-time signals. System calls and functions that can generate signals include raise, kill, killpg, pthread\_kill, tgkill, and sigqueue.
  + **Delivery**: A signal is said to be *pending* until it's delivered. Normally a signal is delivered to a process as soon as possible by the kernel. However, if the process has blocked the signal, it will remain pending until unblocked.
  + **Processing**: Once a signal is delivered, it is processed in one of many ways. Every signal has an associated default action: ignore the signal; or terminate the process, sometimes with a core dump; or stop/continue the process. For non-default behaviour, *handler function* can be called. Exactly which of these happens is specified via sigaction function.

1. Could you explain the concept of blocking and unblocking signals?

Signals remain pending until they are unblocked for delivery. Source: Lynx 2019.

Signals interrupt the normal flow of program execution. This is undesirable when the process is executing some critical code or updating data that's shared with signal handlers. Blocking solves this problem. The tradeoff is that signal handling is delayed.

Every process can specify if it wants to block a specific signal. If blocked and the signal does occur, the operating system will hold the signal as pending. The signal will be delivered once the process unblocks it. The set of currently blocked signals is called **signal mask**.

There's no point blocking a signal indefinitely. For this purpose, the process can instead ignore the signal once it's delivered.

A signal blocked by one processes doesn't affect other processes, who can receive the signal normally.

Signal mask can be set using sigprocmask (single-threaded) or pthread\_sigmask (multi-threaded).  When a process has multiple threads, a signal can be blocked on a per thread basis. Signal will be delivered to any one thread that hasn't blocked it. Essentially,

Signal handlers are per process, signal masks are per thread.

1. Can we have more than one signal pending for a process?

Yes, many standard signals can be pending for a process. However, only one instance of a given signal type can be pending. This is because pending and blocking of signals are implemented as bitmasks, with one bit per signal type. For example, we can have SIGALRM and SIGTERM pending at the same time but we can't have two SIGALRM signals pending. The process will receive only one SIGALRM signal even if raised multiple times.

With real-time signals, signals can be queued along with data,  so that each instance of the signal can individually delivered and handled.

POSIX doesn't specify the order of delivery for standard signals, or what happens if both standard and real-time signals are pending. Linux gives priority to standard signals. For real-time signals, lower numbered signals are delivered first, and if many are queued for a signal type, earliest one is delivered first.

SAMPLE CODE

*// Adapted from source: http://www.firmcodes.com/signals-in-linux/*

*// Accessed: 2019-07-09*

*// Example shows a custom handler for SIGINT*

*// but the handler reverts to default action for future signals.*

*// Thus, first ctrl+c will allow program to continue*

*// and second ctrl+c will terminate the program.*

#include <unistd.h>

#include <stdio.h>

#include <signal.h>

void sig\_handler1(int num)

{

printf("You are here becoz of signal: %d**\n**", num);

signal(SIGINT, SIG\_DFL);

}

int main()

{

signal(SIGINT, sig\_handler1);

while(1)

{

printf("Hello**\n**");

sleep(2);

}

}

## REFERENCES

1. [A Cloud Chef. 2018. "Linux Process States and Signals." Medium, August 27. Accessed 2019-07-09.](https://medium.com/@cloudchef/linux-process-states-and-signals-a967d18fab64)
2. [Bar, Moshe. 2000. "The Linux Signals Handling Model." Linux Journal, May 01. Accessed 2019-07-09.](https://www.linuxjournal.com/article/3985)
3. [Brown, Chris. 2015. "Core Technology: Signals." Linux Voice, November 18. Accessed 2019-07-09.](https://www.linuxvoice.com/core-technology-signals/)
4. [Choudhary, Ajay. 2018. "kill command in Linux with Examples." GeeksforGeeks, December 21. Updated 2019-05-22. Accessed 2019-07-09.](https://www.geeksforgeeks.org/kill-command-in-linux-with-examples/)
5. [FSF. 2019. "Process Signal Mask." Section 24.7.3 in The GNU C Library, glibc 2.29, Free Software Foundation. Updated 2019-02-01. Accessed 2019-07-09.](https://www.gnu.org/software/libc/manual/html_node/Process-Signal-Mask.html)
6. [Fusco, John. 2007. "The Linux Programmer's Toolbox." Pearson Education. Accessed 2019-07-09.](https://books.google.co.in/books?id=5fgn7Tg4Lt8C)
7. [Haim, Liran Ben. 2017. "Linux – Handling Signals in a Multithreaded Application." Developers Area, November 30. Accessed 2019-07-09.](https://devarea.com/linux-handling-signals-in-a-multithreaded-application/)
8. [Hong, K. 2018. "Linux Processes and Signals." BogoToBogo. Accessed 2019-07-09.](https://www.bogotobogo.com/Linux/linux_process_and_signals.php)
9. [Ingargiola, Giorgio. 2009. "Unix Signals." CIS4307 Introduction to Distributed Systems and Networks, College of Science and Technology, Temple University. Accessed 2019-07-09.](https://cis.temple.edu/~ingargio/cis307/readings/signals.html)
10. [Kernel. 1999. "Index of /pub/linux/kernel/v2.2/" Accessed 2019-07-09.](https://mirrors.edge.kernel.org/pub/linux/kernel/v2.2/)
11. [Kerrisk, Michael. 2008. "signal(7) - overview of signals." Linux Programmer's Manual, Linux Foundation. Accessed 2019-07-09.](http://man7.org/linux/man-pages/man7/signal.7.html)
12. [Lynx. 2019. "POSIX Signals for Embedded and Real-Time Developers." Lynx Software Technologies. Accessed 2019-07-09.](https://www.lynx.com/posix-signals-for-embedded-and-real-time-developers/)
13. [Marek. 2012. "Linux process states." Idea of the day, December 11. Accessed 2019-07-09.](https://idea.popcount.org/2012-12-11-linux-process-states/)
14. [Stevens, W. Richard and Stephen A. Rago. 2013. "Advanced Programming in the UNIX® Environment." Third Edition, Addison-Wesley, Pearson Education, Inc. Accessed 2019-07-09.](https://zodml.org/sites/default/files/Advanced_Programming_in_the_UNIX_Environment%2C_3rd_Edition.pdf)
15. [Walberg, Sean, and Ross Brunson. 2015. "CompTIA® Linux+ / LPIC-1 Cert Guide: (Exams LX0-103 & LX0-104/101-400 & 102-400)." Pearson IT Certification, December. Accessed 2019-07-09.](https://www.oreilly.com/library/view/comptia-linux/9780134085777/)
16. [Wikipedia. 2019. "POSIX." Wikipedia, June 25. Accessed 2019-07-09.](https://en.wikipedia.org/wiki/POSIX)
17. [codingfreak. 2009. "Signals in Linux - Blocking Signals." Blog, codingfreak, April 10. Accessed 2019-07-09.](https://codingfreak.blogspot.com/2009/10/signals-in-linux-blocking-signals.html)
18. [vasanth2001. 2008. "How to pass input to a signal handler?" LinuxQuestions.org, August 27. Accessed 2019-07-09.](https://www.linuxquestions.org/questions/programming-9/how-to-pass-input-to-a-signal-handler-665810/)

# **kill command in Linux with Examples**

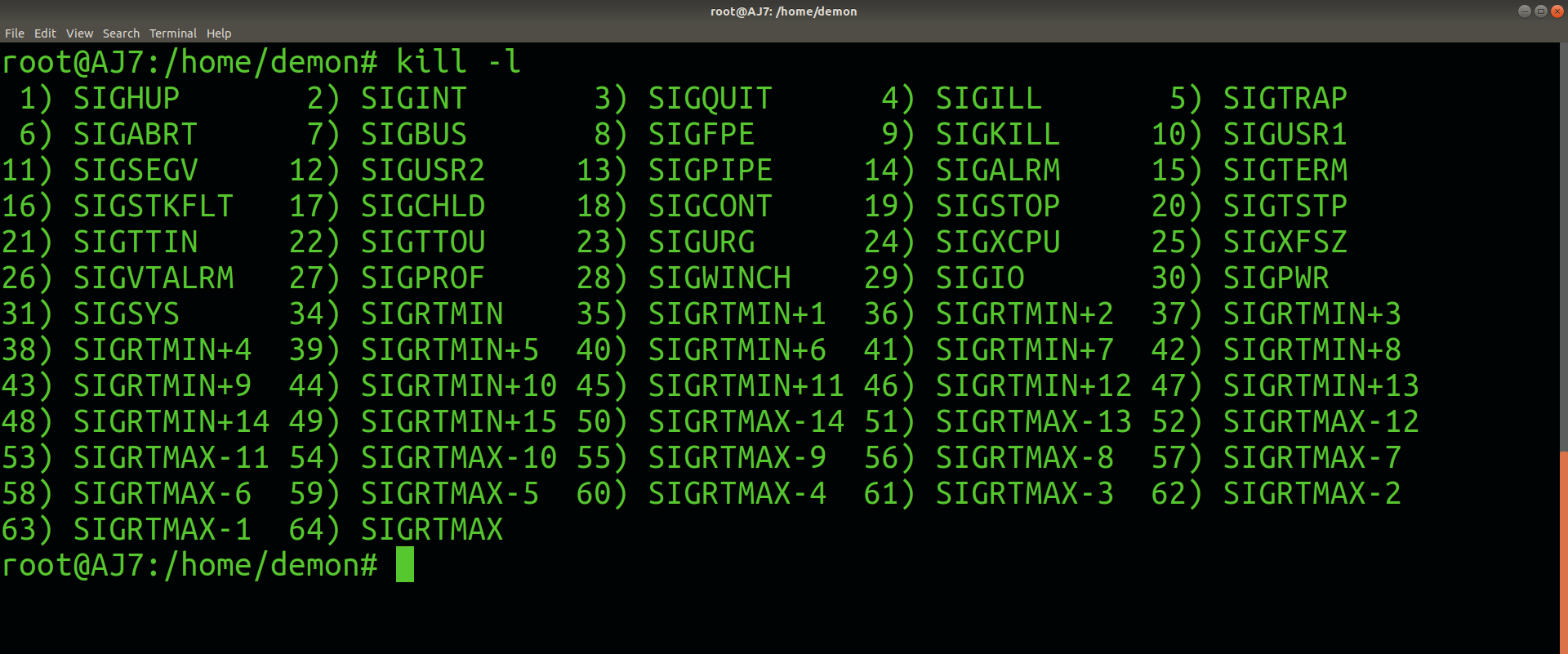
kill command in Linux (located in /bin/kill), is a built-in command which is used to terminate processes manually. kill command sends a signal to a process which terminates the process. If the user doesn’t specify any signal which is to be sent along with kill command then default TERM signal is sent that terminates the process.

#### **Options and examples**

**1. kill -l :**To display all the available signals you can use below command option:

**Syntax:**

$kill -l



Signals can be specified in three ways:

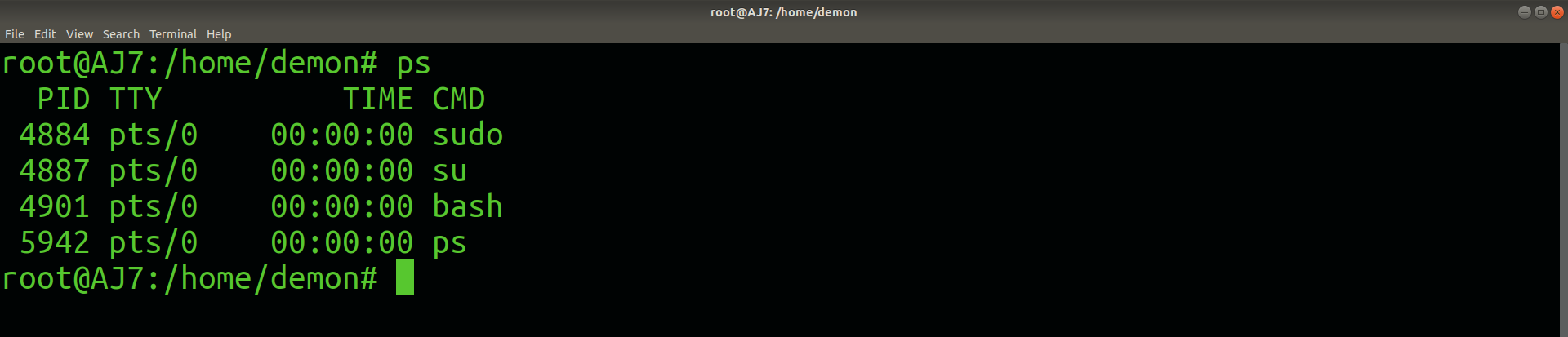
* **By number (e.g. -5)**
* **With SIG prefix (e.g. -SIGkill)**
* **Without SIG prefix (e.g. -kill)**

**Note:**

* Negative PID values are used to indicate the process group ID. If you pass a process group ID then all the process within that group will receive the signal.
* A PID of -1 is very special as it indicates all the processes except kill and init, which is the parent process of all processes on the system.
* To display a list of running processes use the command ps and this will show you running processes with their PID number. To specify which process should receive the kill signal we need to provide the PID.

**Syntax:**

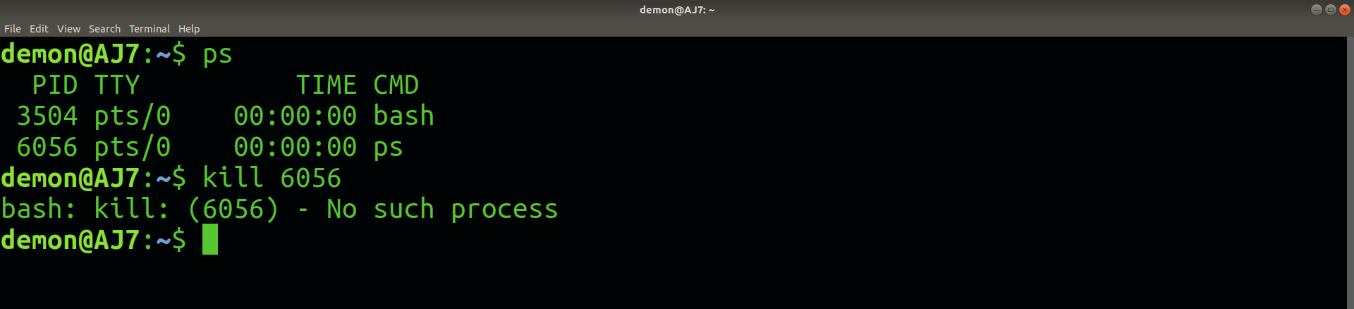
$ps



**2. kill pid :** To show how to use a PID with the kill command.

**Syntax:**

$kill pid



**3. kill -s :** To show how to send signal to processes.

**Syntax:**

kill {-signal | -s signal} pid

**4. kill -L :**This command is used to list available signals in a table format.

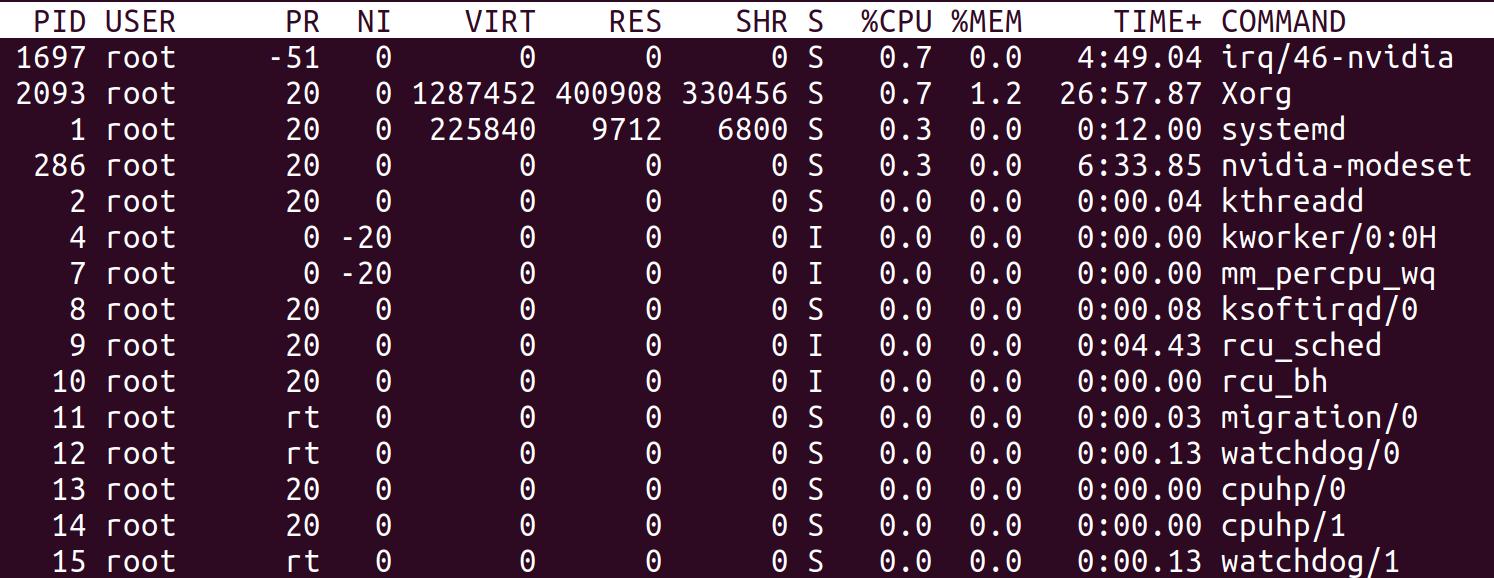
**Syntax:**

kill {-l | --list[=signal] | -L | --table}

# **Linux Process States and Signals**

When troubleshooting a system, it's important to understand the process life-cycle and how the scheduler divides the CPU cores between the running processes and how the kernel communicates with process and how the processes communicate among themselves.

To see the system process states, you can look for the column S of top output (or column STAT of ps x output):

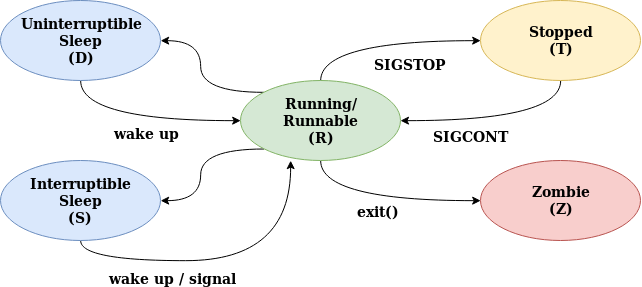


top command output, the process state is the column "s"

Linux has basically 5 states:

* **Running/Runnable (R):** running processes are processes using a CPU core right now, a runnable process is a process that has everything to run and is just waiting for a CPU core slot.
* **Sleeping:** a sleeping process is a process waiting for a resource to be available (for example, a I/O operation to complete) or an event to happen (like a certain amount of time to pass). The difference between process in **Interruptible Sleep (S)** state and **Uninterruptible Sleep (D)** is that the former will wake up to handle signals while the former won't. We'll talk about signals in a moment, but let's suppose that a process is waiting for a I/O operation to complete before wake up. If in the meantime, it receives a signal to terminate (**SIGKILL**), it will terminate before having the chance to handle the requested data. That's why I/O operations normally go to uninterruptible sleep while waiting for the result: they will wake up with when the operation is ready, handle the result and, only then, check for any pending signal to handle. Processes that can be terminated before the wake up condition is fulfilled without any consequence usually go to interruptible sleep instead.
* **Stopped (T):** a process becomes stopped when it receives the **SIGSTOP**signal (like when you press <ctrl>+z in the shell). When stopped, the process execution is suspended and the only signals it will handle are **SIGKILL** and **SIGCONT**. The former will remove the process permanently, while the later will put the process back to the Running/Runnable state (like when you run fg or bg after pressing <ctrl>+z in the shell).
* **Zombie (Z):**we briefly talked about zombie processes when we discussed [system calls](https://medium.com/@cloudchef/linux-system-calls-c2867c7c30c1). When a process finishes with exit() system call, its state needs to be "reaped" by its parent (calling wait()); in the meantime, the child process remains in zombie state (not alive nor dead).

The diagram below helps understand the transition between process states:



# Signals

Signals are one of the ways process communicate among themselves and with the kernel. They can be sent using the system call [**kill**](http://www.tutorialspoint.com/unix_system_calls/kill.htm) (despite the name, it can send any signal, not only **SIGKILL**) and the commands kill and killall.

When receiving a signal, a process can chose to take one of the following actions:

* execute the signal default action
* block the signal setting a signal mask (this is done using the system call **[sigmask](http://www.tutorialspoint.com/unix_system_calls/sigprocmask.htm" \t "_blank)**)
* assign a custom handler to the signal, executing a custom action (using the system call [**signal**](http://www.tutorialspoint.com/unix_system_calls/signal.htm))

Exceptionally, **SIGKILL** and **SIGSTOP** signals cannot be handled or blocked.

The list of the most commonly used signals follow:

* **SIGTERM:** surprisingly, the default signal sent by kill command. Asks the process to terminate voluntarily.
* **SIGKILL:** unlike **SIGTERM,** forces the process to terminate. Can't be blocked or handled.
* **SIGSTOP:**suspend the process execution, putting it in stopped state. In this state, the process will do nothing but accept **SIGKILL** and **SIGCONT** signals. Can be triggered typing <ctrl>+z in the terminal, can’t be blocked or handled.
* **SIGCONT:** if a process is in stopped state, it will put it back in the running/runnable state and resume it execution. If the process is in any other state, it's silently ignored.
* **SIGINT:**generated when the user type <ctrl>+c in the terminal. It interrupts the current command processing and wait for user's next command.
* **SIGQUIT:** generated when the user type <ctrl>+\ in the terminal. Normally, it will force the process to produce a [core dump](https://en.wikipedia.org/wiki/Core_dump) and terminate.
* **SIGALRM:** signal used to wake up sleeping process, normally scheduled by [**alarm**](http://www.tutorialspoint.com/unix_system_calls/alarm.htm) system call.
* **SIGCHLD:** signal send from a child process to its parent process when its state changes . The system call [**wait**](http://www.tutorialspoint.com/unix_system_calls/wait.htm) creates a signal handler for **SIGCHLD** in the parent process; by default it will trigger only when the child calls [**exit**](http://www.tutorialspoint.com/unix_system_calls/exit.htm), but it can be configured to be triggered by another state transitions as well.
* **SIGWINCH:** generated when the terminal detects a change on its size. For full-screen terminal applications, it can trigger a refresh, otherwise can be safely ignored.
* **SIGHUP:** this signal indicates the terminal handling the process has been disconnected and/or the parent process terminated. If you want to run a process that won't terminate when the terminal disconnects, you can start it using the nohup command. Some daemons repurpose this signal to trigger a configuration reload without stopping its execution.
* **SIGUSR1, SIGUSR2:** these signals are reserved for implementing custom actions.