The Linux kill command

Written by: Yoel Monsalve

Some time ago, I heard someone say that "the `kill` command is to kill a process". That is not technically correct. Strictly talking, the `kill` command is to *send a signal to a process*. This is a frequent error I've heard from novice programmers, or people that only touch topics superficially. But, from the Linux help:

```
$ kill --help

kill: kill [-s sigspec | -n signum | -sigspec] pid | jobspec ... or kill -l [sigspec]
```

```
    Send a signal to a job.

    Send the processes identified by PID or JOBSPEC the signal named by
    SIGSPEC or SIGNUM.  If neither SIGSPEC nor SIGNUM is present, then
    SIGTERM is assumed.

    Options:
      -s sig    SIG is a signal name
      -n sig    SIG is a signal number
      -l    list the signal names; if arguments follow `-l' they are
            assumed to be signal numbers for which names should be listed
      -L    synonym for -l

    Kill is a shell builtin for two reasons: it allows job IDs to be used
    instead of process IDs, and allows processes to be killed if the limit
    on processes that you can create is reached.

    Exit Status:
    Returns success unless an invalid option is given or an error occurs.
```

So, the answer is very clear: 'Killing'/terminating a process is only one of the many things you can do through this command. `Signals` are a fascinating topic, and you can know more about it by reading this manpage, or this *computerhope* article, among others.

`Signaling` is a typical and fast communication mechanism between processes. We deliver a signal to a process when we expect such a process to do some specific action. These actions can be, for example, pausing/resuming the process (SIGSTOP, SIGCONT), or terminating it [either in a *nice* or rude way] (SIGTERM,SIGKILL). Some other behaviours are cumbersome, for example, for a parent process to be aware of the termination of one of its children (SIGCHILD), or for asking a process to perform some predefined user-specific action (SIGUSR1). See the complete table of common signal's behaviours on the *computerhope* article.

It is worth mentioning that *signals* are crucial in any well-designed system which is intended to work over a multi-process architecture.

You can see a list of all available signals on your system, by typing

```
$ kill -l

 1) SIGHUP   2) SIGINT   3) SIGQUIT  4) SIGILL   5) SIGTRAP
 6) SIGABRT  7) SIGBUS   8) SIGFPE   9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG  24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF 28) SIGWINCH    29) SIGIO   30) SIGPWR
31) SIGSYS  34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

The syntax for sending a signal to a process, namely the nice termination request signal SIGTERM, is:

```
$ kill -s SIGTERM <pid>
```

where <pid> is the process ID, a unique, integer number that serves to process identification. Each process has an ID which is automatically assigned by the operating system (never try changing it). There are some commands that are useful to inquire the PID of a program, like pgrep, or ps. However, that is enough matter for another article and will not be covered here.

Other ways for the kill syntax are:

```
$ kill -s TERM <pid>
```

and (a kind of shortcut):

```
$ kill -TERM <pid>
```

or, by using the signal *number* instead of name (**NOT recommended**, as numbers might hypothetically vary in the future, or across systems):

```
$ kill -n 15 <pid>
```

```
$ kill 15 <pid>
```

A first experiment

Suppose that we have a "*lazy*" process which does not do another thing than sleeping for a while. Example:

```python
"""FILE: lazy.py
"""
from time import sleep
import os

print('I\'m a very lazy process -.). Please, don\'t disturb ...')
print(f'My PID = {os.getpid()}')
sleep(10)
print('Lazy program says ... Bye !\n')
```

The interesting thing is that once our lazy process is sleeping and annoying our console, we can just stop it, by typing the shortcut `CTRL + Z` (on UNIX-like systems). The process is then stopped:

```
$ python lazy.py
I'm a very lazy process -.). Please, don't disturb ...
My PID = 76750
^Z
[1]+  Stopped                 python lazy.py
```

Notice that the program is printing its own PID, in order to facilitate you sending it a signal, when you need to.

Now, once we have time again to watch our lazy process, let's just resume it, by delivering the signal `SIGCONT` (continue):

```
$ kill -CONT 76750
$ Lazy program says ... Bye !
```

Now, after a while, the program terminates and prints out a goodbye message.

This stopping mechanism can be used, for example, to take 'snapshots' of processes that direct its output to the console.

Getting familiarized with *jobs*

Working on UNIX-like terminal can be productive when it comes to using *jobs*. A job is a process running under a terminal. You could have noted the particular message

```
^Z
[1]+  Stopped                 python lazy.py
```

when we stopped the lazy program. This means that the current terminal is identifying such a process with the *job number* equal to 1 (it might be a higher number if you have several jobs running on the same terminal).

According to the `kill` help, it is also possible to use the command with the syntax:

```
kill [-s sigspec | -n signum | -sigspec] jobspec
```

where `jobspec` is the identification for the target process (including a '%' in front of the number). Namely:

```
$ python lazy.py
I'm a very lazy process -.). Please, don't disturb ...
My PID = 77330
^Z                             <-- you press CTRL + Z
[1]+  Stopped                 python lazy.py

$ kill -CONT %1                <-- this will resume process
$ Lazy program says ... Bye !    <-- process is done
```

The easy way

If you just want to pause/resume a process, you can go further into the concept of *jobs*. Pressing `CTRL + Z` puts the job in paused state, i.e., in *background*. We can later reactivate the paused process, by bringing it into *foreground*.

STATE	**MEANING**
background	put the process in paused state
foreground	awake the process and enable it for interacting with keyboard

We can move a process to the background (pausing) or foreground (resuming) by using, respectively, the commands fg and bg. Example

```
fg %1    .... puts the job ID=1 into background state (stop)
bg %1    .... puts the job ID=1 into foreground state (resume)
```

You can switch a process back and forth between background and foreground as many times as you want, while the process is still alive and being owned by that terminal.

Pressing `CTRL + Z` implicitly sends a `STOP` signal to a process, so it is already put into the background state. To restore it, use `fg`. You can later move it to the background with `bg`, then again to the foreground with `fg`, ... etc.

```
$ python lazy.py
I'm a very lazy process -.). Please, don't disturb ...
My PID = 77771
^Z                                <-- you press CTRL + Z
[1]+  Stopped                  python lazy.py
$ fg %1                           <-- this will resume child, until its completion
python lazy.py
Lazy program says ... Bye !
```

Notice that `fg` implicitly sends a `CONT` signal to the child, so using `fg`/`bg` is almost equivalent to using the manual way of `kill -STOP|-CONT`.

Exploring other options

Maybe you decided that you have had enough of this lazy process, and you want to simply terminate it. Then, you dispatch the `TERM` signal to end up the annoying thing.

```
$ python lazy.py
I'm a very lazy process -.). Please, don't disturb ...
My PID = 77907
^Z                                <-- you press CTRL + Z
[1]+  Stopped                  python lazy.py
$ kill -TERM %1                   <-- kill the annoying bug

[1]+  Stopped                  python lazy.py
$                                 <-- press ENTER again, and you will see
[1]+  Terminated               python lazy.py
                                  (the annoying thing is gone!)
```

OK, this can look a bit strange. Once you just deliver the termination signal, the console responds telling that the process is stopped. Press ENTER again and the console will inform that the child has been killed. That is just one of those UNIX-behaviours (and can be modified by setting some shell variables, but we won't deepen into it by now.)

A final note: The `TERM` signal sends somehow a kindly termination request to the target. It is supposed to perform some "on-close" procedures, like closing open file descriptors, propagating the signal to their own children, and so on. There also be the other rude signal named `KILL` that can't be ignored and closes the target program immediately and unconditionally. But, be aware, only do things like this:

```
$ kill -KILL <pid>|<jobspec>
```

as a last resource. The ideal, is to first send `TERM`, then after a few seconds, try `TERM` again, and after some fruitless attempts, send the `KILL` signal.

And, that's all for now, hoping that you enjoyed this material and expect to share more articles soon !