## Insider Tips: Signal-Based Inter-Process Communication

# Hand-Signals

A typical Unix computer runs at least 30 simultaneous processes. Processes often need to communicate. Signals provide the best-known method for doing so. Users and admins can use the *kill* command to issue signals.

BY MARC ANDRÉ SELIG

A process is called, completes its job, cleans up, and terminates. If the job is more complex, this can take a while – weeks, or even months. Take a Web server, for example, which will typically reside in memory until the next update is released. This applies to most daemons. That does not mean to say that the environment for this kind of software will always be the same. After modifying a configuration file, an admin may need to draw a daemon's attention to the changes. This typically involves inter-process communication.

## Signals

Signals are one of the best-known IPC (inter-process communication) techniques. Each time you enter the *kill* command to terminate an unneeded process, you are actually sending a signal to that process (see Figure 1). Figure 2 shows a list of signals defined for *kill*. The list in the */usr/include/bits/signum.h* header file is authoritative for programs.

If you attempt to kill a process without supplying any additional parameters, your command will assume signal 15 (*TERM*, terminate). However, some processes simply ignore this signal, or may already have crashed. In this case, the *TERM* signal is no use. To obtain the desired results, you will need to type *kill -9* instead, thus transmitting a *KILL* signal. Unless you happen to have chosen a zombie (see box - Zombies), or a process that is waiting for an I/O operation, and thus residing in the kernel space, the process is bound to be terminated.

## Stopping Programs

Some of the other signals are very important – although this may not be so obvious at first glance. Most readers will have pressed [Ctrl] + [Z] to interrupt an editor. The shell reacts by outputting:

```
[2]+  Stopped       vi myfile
```

You can reinstate the editor later by typing *fg* (foreground). The *jobs* command tells you the status of the processes currently controlled by the shell:

```
[1]-  Running       emacs &
[2]+  Stopped       vi myfile
```

[Ctrl] + [Z] sends signal 19, *STOP*. The *fg* command continues the interrupted processes by issuing signal 18, *CONT* (continue). Signal 18 also issues the *bg* command, which sends the process into the background, just like adding *&* while calling a process. If you temporarily stop a background process (because it is consuming too many resources, for example), and want to continue the process later, you can use *kill -19* to stop, and *kill -18* to continue the process.

The previous example demonstrates a simplified notation for process numbers, which bash offers as a shortcut. Normally, each Linux process has a PID (process ID) between 1 and 32767, where 1 is reserved for *init*, the granddaddy of all processes. The first few hundred PIDs are used by the kernel's own processes. The remaining process IDs are used by userspace programs.

It is tiring, repeatedly typing *kill -18 21967* without any typos. Bash provides two simplifications. First, it knows the PIDs of the current and previous jobs. These are tagged with plus and minus respectively in the job list. If

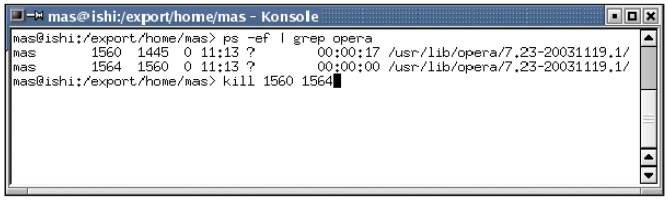


**Figure 1: The user *mas* runs *ps* and *grep* to search for Opera processes and then executes *kill* PID to tell those processes to terminate.**

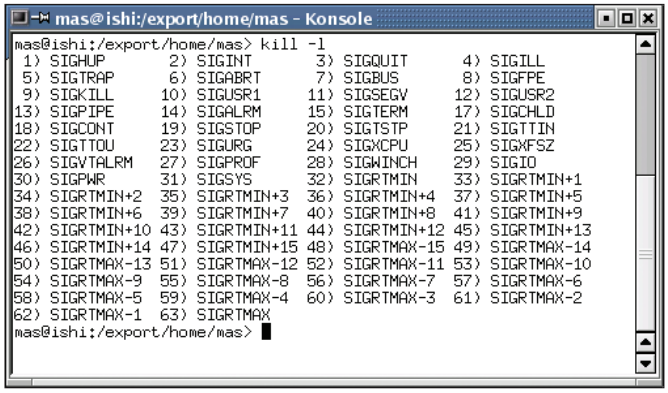


**Figure 2: Linux recognizes 64 different signals which have both a name and a number. The frequently-used *kill -9* command can also be expressed as *kill -SIGKILL* or *kill -KILL*.**

you use a shell command like *fg* to manipulate a process, but do not specify a PID, the command is applied to the current process. The second thing that bash does is to provide process name shortcuts, *[1]* and *[2]* in this case. You can prefix a percent character to the shortcut to access these processes. *fg %1* in the previous example would send Emacs to the foreground.

In addition to [Ctrl] + [Z] (suspend), some shells also honor a delayed suspend, [Ctrl] + [Y]. While suspend immediately sends a *STOP* signal, [Ctrl] + [Y] waits until the process reads data from the terminal. This allows a process to be manipulated immediately after accepting input.

## Hangups

Another signal dates back to the days of serial terminals. Signal number 1 is called *HUP* or hangup. If you use a modem to access a Unix system and the telephone line goes down, your shell will be sent a *HUP* signal and can clean up. For example, an editor would create a backup and then terminate.

Modem lines are uncommon today, but the *HUP* signal retains its usefulness. Terminating a SSH connection is one common use, and a common pitfall.

Imagine the scenario from Figure 3a. User *mas* moves to a new machine to perform some remote management tasks and launches a time-consuming job. As he does not want to wait for the results, he appends an ampersand (*&*) to send the process into the background, before closing the session. Unfortunately, this does not seem to work; *ssh* appears to hang. The user sees no alternative but to press ~., the SSH escape sequence, to kill the connection.

### Zombies

A process that spawns another is referred to as a parent process, and the relationship between the two as a parent/child relationship. The calling process is the parent, and the called process the child. If the child process terminates before its parent, the parent is sent a *CLD* or *CHLD* (child) signal. The parent is expected to confirm the demise of its child, so to speak. If the parent fails to do so, the child process is left in the process table as a zombie [1]. After all, how is the kernel to know if or when the parent will check the exit status of its child? The process entry has to be kept, to cover this possibility.

Most of the other resources will have been released by this time. The former child no longer exists, and *kill -9* will have no effect on it. The zombie disappears when the parent process terminates or queries the exit status of the child.

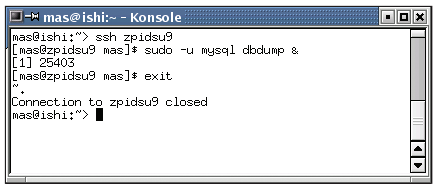


**Figure 3a: SSH waits for child processes to terminate before closing the connection. Typing ~. kills the connection.**

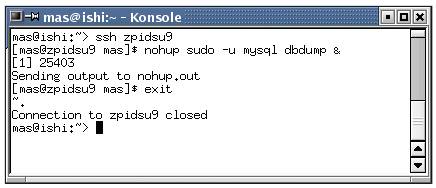


**Figure 3b: Time-consuming background processes need to be launched with *nohup*, if they are to survive the ~. escape sequence.**

What has actually happened is that SSH has not crashed, but simply noticed the background task. SSH was waiting for this process to terminate before closing the connection. When the connection is killed, a *HUP* signal is sent to the background process, thus terminating the process.

Figure 3b shows the right way to do this. The *nohup* command screens a background process from the hangup signal. SSH notices this and will again start to wait, but this time the connection can be killed without affecting the background process. *dbdump* keeps on running, and will proudly present its results the next time *mas* logs on. This *dbdump* syntax is more elegant:

```
nohup sudo -u mysql dbdump ➥
</dev/null &
```

This means that the process is not connected to the terminal and SSH will quit after the user logs out, without having to type ~. to break the connection.

## Modified Configurations

The convention is to use *HUP* to tell the daemon about modifications to its configuration file.

The syslog daemon [2] is a typical example. syslog is normally expected to log warnings and error messages, saving hard disk capacity and the administrator from headaches. A little more is needed after installing a new software package. If the software does not run as expected, the admin will need additional messages and debugging information. By adding a line to */etc/syslog.conf* the admin can tell syslog to collect this data:

```
*.*    -/var/log/everything
```

You need to send a *HUP* signal to let *syslogd* know about the change. *ps -ef | grep syslog* reveals the PID for the syslog daemon, and *kill -1* PID takes care of terminating the daemon. If you do not have time to locate the PID, some distributions provide a *killall* which allows you to send a signal to all processes with the name you supply:

```
killall -1 syslogd
```

## User-Defined Signals

Some other signals, such as, 10 (*USR1*), and 12 (*USR2*), are available for user-defined tasks. If you modify the Apache configuration file, you can force the Apache daemon to re-parse the file by sending it a *HUP* signal. However, this means dropping any current jobs. If you send a *USR1* signal instead, the Apache parent process waits until the current jobs have finished before stopping and restarting its child processes.

Signals are a highly widespread, and important part of an admin's daily life. There are a few restrictions. The number of signals is limited. For safety reasons, only the root user is allowed to send signals to processes that do not belong to his user account.

This justifies alternative mechanisms. We will be looking at the subject in a future issue of Admin Workshop. ■

### INFO

[1] Wikipedia entry on zombie processes: *http://en.wikipedia.org/wiki/Zombie_process*

[2] Marc André Selig, "The System Logger", Linux Magazine, Issue 40, March 2004, p64.