Processes in Linux/Unix

A program/command when executed, a special instance is provided by the system to the process. This instance consists of all the services/resources that may be utilized by the process under execution.

- Whenever a command is issued in unix/linux, it creates/starts a new process. For example, pwd when issued which is used to list the current directory location the user is in, a process starts.
- Through a 5 digit ID number unix/linux keeps account of the processes, this number is call process id or pid. Each process in the system has a unique pid.
- Used up pid's can be used in again for a newer process since all the possible combinations are used.
- At any point of time, no two processes with the same pid exist in the system because it is the pid that Unix uses to track each process.

Initializing a process

A process can be run in two ways:

1. **Foreground Process :** Every process when started runs in foreground by default, receives input from the keyboard and sends output to the screen.
   When issuing pwd command

   ```
   $ ls pwd
   ```

   Output:

   ```
   $ /home/geeksforgeeks/root
   ```

   When a command/process is running in the foreground and is taking a lot of time, no other processes can be run or started because the prompt would not be available until the program finishes processing and comes out.

2. **Backround Process :** It runs in the background without keyboard input and waits till keyboard input is required. Thus, other processes can be done in parallel with the process running in background since they do not have to wait for the previous process to be completed.
   Adding & along with the command starts it as a background process

3. **$ pwd &**

Since pwd does not wants any input from the keyboard, it goes to the stop state until moved to the foreground and given any data input. Thus, on pressing Enter, :
Output:

```
[1]   +   Done                  pwd
$
```

That first line contains information about the background process – the job number and the process ID. It tells you that the ls command background process finishes successfully. The se The second is a prompt for another command.

Tracking ongoing processes

ps (Process status) can be used to see/list all the running processes.

```
$ ps

PID        TTY       TIME         CMD
19         pts/1     00:00:00     sh
24         pts/1     00:00:00     ps
```

For more information -f (full) can be used along with ps

```
$ ps –f

UID       PID  PPID C STIME    TTY         TIME CMD
52471      19     1 0 07:20    pts/1   00:00:00f     sh
52471      25    19 0 08:04    pts/1   00:00:00      ps -f
```

For a single process information, ps along with process id is used

```
$ ps 19

PID        TTY       TIME         CMD
19         pts/1     00:00:00     sh
```

For a running program (named process) **Pidof** finds the process id's (pids)

Fields described by ps are described as:
UID: User ID that this process belongs to (the person running it)
PID: Process ID
PPID: Parent process ID (the ID of the process that started it)
C: CPU utilization of process
STIME: Process start time
TTY: Terminal type associated with the process
TIME: CPU time taken by the process
CMD: The command that started this process

There are other options which can be used along with ps command :
-a: Shows information about all users
-x: Shows information about processes without terminals
-u: Shows additional information like -f option
-e: Displays extended information

Stopping a process

When running in foreground, hitting Ctrl + c (interrupt character) will exit the command. For processes running in background kill command can be used if it's pid is known.

```
$ ps –f

UID        PID  PPID C STIME    TTY          TIME CMD
52471       19     1 0 07:20    pts/1  00:00:00      sh
52471       25    19 0 08:04    pts/1  00:00:00      ps –f

$ kill 19
Terminated
```

If a process ignores a regular kill command, you can use kill -9 followed by the process ID .

```
$ kill –9 19
Terminated
```

Other process commands:

bg: A job control command that resumes suspended jobs while keeping them running in the background
Syntax:

```
bg [ job ]
```

For example

```
bg %19
```

fg: It continues a stopped job by running it in the foreground.
Syntax:

```
fg [ %job_id ]
```

For example

```
fg 19
```

top: This command is used to show all the running processes within the working environment of Linux.
Syntax:

```
top
```

nice: It starts a new process (job) and assigns it a priority (nice) value at the same time.
Syntax:

```
nice [-nice value]
```

nice value ranges from -20 to 19, where -20 is of the highest priority.

renice : To change the priority of an already running process renice is used.
Syntax:

```
renice [-nice value] [process id]
```

df: It shows the amount of available disk space being used by file systems
Syntax:

```
df
```

Output:

```
Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/loop0      18761008  15246876   2554440  86% /
none                   4         0         4   0% /sys/fs/cgroup
udev              493812         4    493808   1% /dev
tmpfs             100672      1364     99308   2% /run
none                5120         0      5120   0% /run/lock
none              503352      1764    501588   1% /run/shm
none              102400        20    102380   1% /run/user
/dev/sda3      174766076 164417964  10348112  95% /host
```

free: It shows the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel
Syntax:

```
free
```

Output:

```
             total       used       free     shared    buffers     cached
Mem:       1006708     935872      70836          0     148244     346656
-/+ buffers/cache:     440972     565736
Swap:       262140     130084     132056
```

Types of Processes

1. **Parent and Child process :** The 2nd and 3rd column of the ps –f command shows process id and parent's process id number. For each user process there's a parent process in the system, with most of the commands having shell as their parent.

2. **Zombie and Orphan process :** After completing its execution a child process is terminated or killed and SIGCHLD updates the parent process about the termination and thus can continue the task assigned to it. But at times when the parent process is killed before the termination of the child process, the child processes becomes orphan processes, with the parent of all processes "init" process, becomes their new ppid.
   A process which is killed but still shows its entry in the process status or the process table is called a zombie process, they are dead and are not used.
3. **Daemon process :** They are system-related background processes that often run with the permissions of root and services requests from other processes, they most of the time run in the background and wait for processes it can work along with for ex print daemon. When ps –ef is executed, the process with ? in the tty field are daemon processes

Internal and External Commands in Linux

The UNIX system is command-based *i.e* things happen because of the commands that you key in. All UNIX commands are seldom more than four characters long.
They are grouped into two categories:

- **Internal Commands :** Commands which are built into the shell. For all the shell built-in commands, execution of the same is fast in the sense that the shell doesn't have to search the given path for them in the PATH variable and also no process needs to be spawned for executing it.
  Examples: source, cd, fg etc.
- **External Commands :** Commands which aren't built into the shell. When an external command has to be executed, the shell looks for its path given in PATH variable and also a new process has to be spawned and the command gets executed. They are usually located in /bin or /usr/bin. For example, when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.
  Examples: ls, cat etc.

If you know about UNIX commands, you must have heard about the **ls** command. Since **ls** is a program or file having an independent existence in the /bin directory(or /usr/bin), it is branded as an **external command** that actually means that the ls command is not built into the shell and these are executables present in separate file. In simple words, when you will key in the ls command, to be executed it will be found in /bin. Most commands are external in nature, but there are some which are not really found anywhere, and some which are normally not executed even if they are in one of the directories specified by PATH. For instance, take **echo** command:

```
$type echo
echo is a shell builtin
```

echo isn't an external command in the sense that, when you type echo, the shell won't look in its PATH to locate it(even if it is there in /bin). Rather, it will execute it from its own set of built-in commands that are not stored as separate files. These built-in commands, of which **echo** is a member, are known as **internal commands**.

You now might have noticed that it's the shell that actually does all this works. This program starts running as soon the user log in, and dies when the user log out. The shell is an external command with a difference, it possesses its own set of internal commands. So, if a command exists both as an internal command of the shell as well as external one(in /bin or /usr/bib), the shell will accord top priority to its own internal command of the same name.

This is exactly the case with **echo** which is also found in /bin, but rarely ever executed because the shell makes sure that the internal **echo** command takes precedence over the external. Now, talk more about the internal and external commands.

Getting the list of Internal Commands

If you are using bash shell you can get the list of shell built-in commands with **help** command :

```
$help

// this will list all
the shell built-in commands //
```

How to find out whether a command is internal or external?

In addition to this you can also find out about a particular command *i.e* whether it is internal or external with the help of **type** command :

```
$type cat
cat is /bin/cat

//specifying that cat is
external type//

$type cd
cd is a shell builtin

//specifying that cd is
internal type//
```

Internal vs External

The question that when to use which command between internal and external command is of no use cause the user uses a command according to the need of the problem he wants to solve. The only difference that exists between internal and external command is that internal commands

work much faster than the external ones as the shell has to look for the path when it comes to the use of external commands.

There are some cases where you can avoid the use of external by using internal in place of them, like if you need to add two numbers you can do it as:

```
//use of internal command let
for addition//
```

```
$let c=a+b
```

instead of using :

```
//use of external command expr
for addition//
```

```
$c=`expr $a+$b`
```

In such a case, use of let will be more better option as it is a shell built-in command so will work faster than the expr which is an external command.

Process Creation:

In UNIX and POSIX you call **fork()** and then **exec()** to create a process. When you **fork** it clones a copy of your current process, including all data, code, environment variables, and open files. This child process is a duplicate of the parent (except for a few details). **fork()** returns the process ID of the child to the parent process and a zero to the child process (*remember, both are now executing*). The child may then call **exec()** to replace itself with the code of a different program (if that was your goal). The new *child* process will have the same files open as the *parent*, except those whose close-on-exec flag was set with **fcntl**.

In pseudo-code, this is:

```
result := fork()
if result < 0
   #fork() failed
elseif result > 0
   #this is the Parent
elseif
   #result=0, this is the Child
   #call exec() to "replace myself" with another prog
   exec()
   #control will Never return here from exec()
   #instead if goes to the new program code
endif
```

About trap

trap is a function built into the shell that responds to hardware signals and other events.

Description

trap defines and activates handlers to be run when the shell receives signals or other special conditions.

ARG is a command to be read and executed when the shell receives the signal(s) *SIGNAL_SPEC*. If *ARG* is absent (and a single *SIGNAL_SPEC* is supplied) or *ARG* is a dash ("**-**"), each specified signal is reset to its original value. If *ARG* is the null string, each *SIGNAL_SPEC* is ignored by the shell and by the commands it invokes.

If a *SIGNAL_SPEC* is **EXIT** (**0**), *ARG* is executed upon exit from the shell.

If a *SIGNAL_SPEC* is **DEBUG**, *ARG* is executed before every simple command.

If a *SIGNAL_SPEC* is **RETURN**, *ARG* is executed each time a shell function or a script run by the "**.**" or **source** built-in commands finishes executing.

A *SIGNAL_SPEC* of **ERR** means to execute *ARG* each time a command's failure would cause the shell to exit when the **-e** option is enabled.

If no arguments are supplied, **trap** prints the list of commands associated with each signal.

trap syntax

```
trap [-lp] [[ARG] SIGNAL_SPEC...]
```

Options

-l print a list of signal names and their corresponding numbers.

-p display the trap commands associated with each *SIGNAL_SPEC*.

trap and onitr examples

```
trap -l
```

Display a list of signal names and their corresponding numbers. The list will resemble the following:

```
 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
trap
```

Display a list of the currently-set signal traps.

```
trap 'rm -f /tmp/xyz$$; exit' ERR EXIT
```

stty command is used to manipulate the terminal settings. You can view and modify the terminal settings using this command as explained below.

1. Display All Settings

-a option displays all the stty settings in a user friendly readable format as shown below.

```
# stty -a
speed 38400 baud; rows 59; columns 208; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = ; eol2 = ; swtch
= ; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V;
flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts -cdtrdsr
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -
ixoff -iuclc -ixany -imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0
ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
```

You can also speciay -all, which is same as -a.

```
stty --all
```

2. Display All Settings (in Stty format)

The following will display all the settings in a format that is readable by stty. Please note that this is not in readble format. You'll typically use this if you want to transfers the settings between systems.

```
# stty -g
500:5:bf:8a3b:3:1c:7f:15:4:0:1:0:11:13:1a:0:12:f:17:16:0:0:0:0:0:0:0:0:0:0:
0:0:0:0:0
```

You can also use –save option, which is same as -g

```
# stty --save
```

```
500:5:bf:8a3b:3:1c:7f:15:4:0:1:0:11:13:1a:0:12:f:17:16:0:0:0:0:0:0:0:0:0:0:
0:0:0:0:0:0
```

3. Specify Device

You can specify a device file as an argument to stty command. In that case, it will use the device that you've specified instead of using the standard input.

```
# stty -F /dev/pts/0
speed 38400 baud; line = 0;
-brkint -imaxbel
```

4. Set a Stty Value

The following example sets a stty value istrip.

```
# stty istrip
```

As you see below, istrip is set

```
# stty -a | grep istrip
-ignbrk -brkint -ignpar -parmrk -inpck istrip -inlcr -igncr icrnl ixon -ixoff
```

5. Negate a Stty Value

To negate a stty value, you need to specify a – in front of the value. The following example negates the stty value istrip.

```
# stty -istrip
```

As you see below, istrip is negated. i.e there is a – in front of the istrip.

```
# stty -a | grep istrip
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -
ixoff
```

Kill command in UNIX and Linux is normally used to kill a suspended or hanged process or process group. Though kill is mainly associated with kill operation its mere a signal transporter and can send specified signal to specified process in UNIX or UNIX like systems e.g. Linux, Solaris or FreeBSD. Like in windows when we see a particular process hung the system we go to task manager find the process and kill it, similarly in UNIX and Linux we first find the process ID (PID) of offending process and then kill it. Though we have killAll command also which doesn't require PID instead it can kill the process with just process name. **Kill commands** is often a wrapper around kill () system call but some Linux systems also has built-in kill in place. In this article we will see some examples of *kill command in UNIX* and how we can use kill command to kill the locked process.

Read more: https://javarevisited.blogspot.com/2011/12/kill-command-unix-linux-example.html#ixzz5SrVE1Mlj

1) Kill command to forcefully kill a process in UNIX

kill -9 is used to forcefully terminate a process in Unix. Here is syntax of kill command in UNIX.

Read more: https://javarevisited.blogspot.com/2011/12/kill-command-unix-linux-example.html#ixzz5SrVNNbnI

```
ps -ef| grep process  identifier // will give you PID

kill -9 PID
```

2) Unix kills command to kill multiple processes

With kill command in UNIX you can specify multiple PID at same time and all process will be signaled or if signal is KILL they get killed like below kill command in UNIX

Syntax of kill in UNIX for killing multiple processes:

```
kill -9 pid1 pid 2
```

Here is an example of killing multiple processes in UNIX:

```
trader@asia:/ ps -ef

UID    PID   PPID TTY    STIME COMMAND

trader   5736   5332  1   Nov 14 /usr/bin/bash

trader   5604   5552  0   Nov 16 /usr/bin/bash

trader   3508   4872  2   Nov 17 /usr/bin/bash

trader   6532   5604  0  17:43:19 /usr/bin/man

trader   6352   3420  0  17:43:22 /usr/bin/sh

trader   7432   6352  0  17:43:22 /usr/bin/less

trader   5348   3508  2  17:52:59 /usr/bin/ps



trader@asia:/ kill -9 3420 6352



trader@asia:/ ps -ef
```

```
UID    PID   PPID TTY    STIME COMMAND

trader   5736   5332  1   Nov 14 /usr/bin/bash

trader   5604   5552  0   Nov 16 /usr/bin/bash

trader   3508   4872  2   Nov 17 /usr/bin/bash

trader   5040   3508  2  17:53:38 /usr/bin/ps
```

3) Kill command in UNIX to find Signal name

Kill command can also show you name of Signal if you rung it with option "-l". For example "9" is **KILL signal** while "3" is **QUIT signal**.

```
trader@asia:/ kill -l 3

QUIT



trader@asia:/ kill -l 9

KILL
```

4) Printing all signals supported by kill in UNIX

You can use **kill -l** to list down all signals supported by kill command in UNIX as shown in below example:

```
trader:~ 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
```

5) Sending signals using -s option of kill command in UNIX.

Instead of specifying number you can specify name of signal you are sending to other process with kill command option "-s". Here is an example of using Kill command in UNIX with signal code.

```
trader:~ ps -ef

UID   PID   PPID TTY    STIME COMMAND

trader   5736   5332   1   Nov 14 /usr/bin/bash

trader   3508      1   2   Nov 17 /usr/bin/bash

trader   7528   2352   0  18:00:30 /usr/bin/bash

trader   4424   7528   0  18:05:11 /usr/bin/less

trader    168   7528   0  18:05:15 /usr/bin/ps


[1]+  Stopped                 less -r a


trader:~ kill -s KILL 4424


trader:~ ps -ef

UID   PID   PPID TTY    STIME COMMAND

trader   5736   5332   1   Nov 14 /usr/bin/bash

trader   3508      1   2   Nov 17 /usr/bin/bash

trader   7528   2352   0  18:00:30 /usr/bin/bash

trader   5044   7528   0  18:05:32 /usr/bin/ps

[1]+  Killed                  less -r a
```

Important point about kill command in UNIX and Linux

To summarize discussion and examples of **UNIX kill command**, I have outlined some of the important points and things to remember related to kill command in UNIX and Linux. You can quickly refer this point whenever you have some doubt over kill in UNIX.

1) *Kill command in UNIX* can send signals to any other process in UNIX or Linux.In order to work with those signals corresponding process should understand those signals.

2) You can get full list of signals supported by *kill command in unix* is by simply doing "man kill" or simply by executing command **kill -l**.

3) Bash has a built-in kill routine. So you can check that by typing **/bin/kill –version**

Read more: https://javarevisited.blogspot.com/2011/12/kill-command-unix-linux-example.html#ixzz5SrVU8jL3

Unix job control command list

The following table lists the basic Unix job control commands:

Command	Explanation	Example
`&`	Run the command in the background	`% long_cmd &`
`Ctrl-z`	Stop the foreground process	`[Ctrl-z] Stopped`
`jobs`	List background processes	`% jobs [1] - Stopped vi [2] - big_job &`
`%n`	Refers to the background number `n`	`% fg %1`
`%?str`	Refers to the background job containing `str`	`% fg %?ls`
`bg`	Restart a stopped background process	`% bg [2] big_job &`
`fg`	Bring a background process to the foreground	`% fg %1`
`kill`	Kill a process	`% kill %2`
`~ Ctrl-z`	Suspend an `rlogin` or ssh session	`host2>~[Ctrl-z] Stopped host1>`
`~~ Ctrl-z`	Suspend a second level `rlogin` or `ssh` session	`host3>~~[Ctrl-z] Stopped host2>`

What is a Process?

An instance of a program is called a Process. In simple terms, any command that you give to your Linux machine starts a new process.

Having multiple processes for the same program is possible.

Types of Processes:

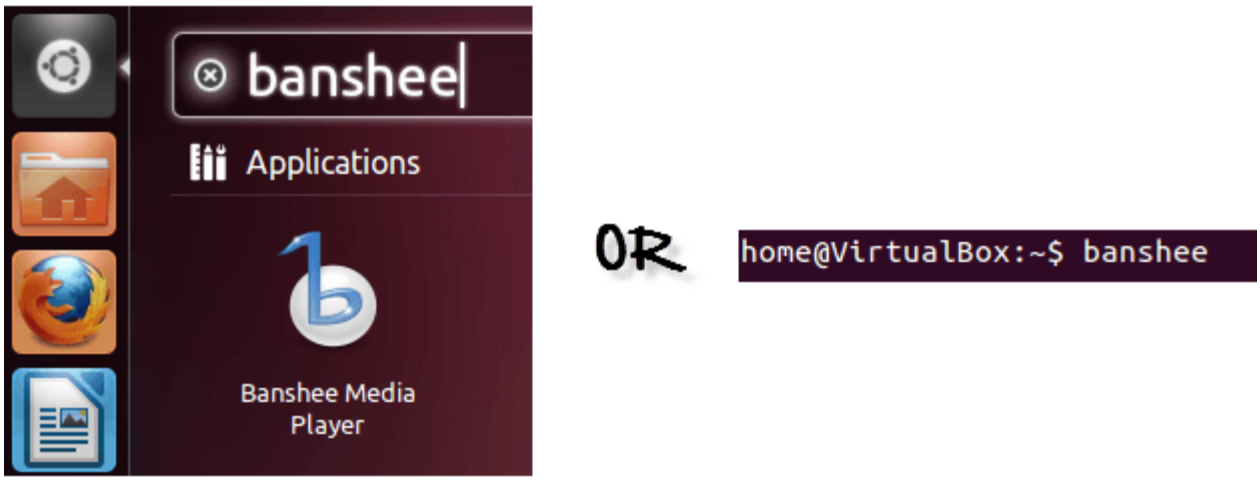
- Foreground Processes: They run on the screen and need input from the user. For example Office Programs
- Background Processes: They run in the background and usually do not need user input. For example Antivirus.

Please be patient. The Video will load in some time. If you still face issue viewing video click here

Running a Foreground Process

To start a foreground process, you can either run it from the dashboard, or you can run it from the terminal.

When using the Terminal, you will have to wait, until the foreground process runs.

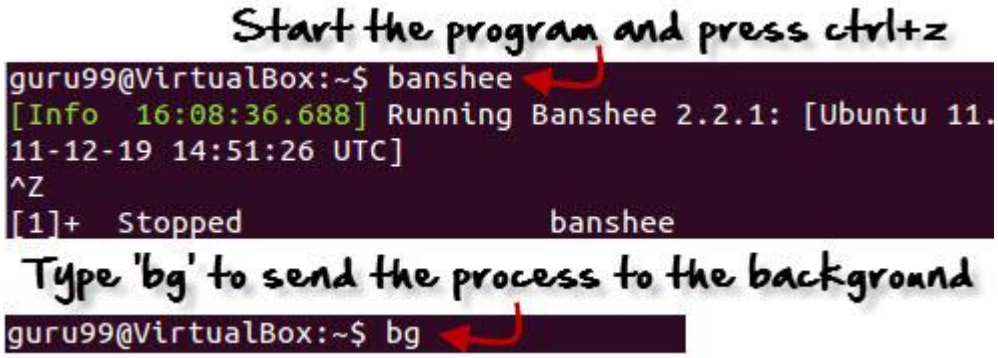


Running a Background process

If you start a foreground program/process from the terminal, then you cannot work on the terminal, till the program is up and running.

Particular, data-intensive tasks take lots of processing power and may even take hours to complete. You do not want your terminal to be held up for such a long time.

To avoid such a situation, you can run the program and send it to the background so that terminal remains available to you. Let's learn how to do this -



Fg

You can use the command "fg" to continue a program which was stopped and bring it to the foreground.

The simple syntax for this utility is:

```
fg jobname
```

Example

1. Launch 'banshee' music player
2. Stop it with the 'ctrl +z' command
3. Continue it with the 'fg' utility.

```
home@VirtualBox:~$ banshee
^Z
[1]+  Stopped                 banshee
home@VirtualBox:~$ fg banshee
banshee
[Info  00:36:19.400] Running Banshee 2.2.0: [Ubuntu oneiric
 (linux-gnu, i686) @ 2011-09-23 04:51:00 UTC]
```

Let's look at other important commands to manage processes -

Top

This utility tells the user about all the running processes on the Linux machine.

```
home@VirtualBox:~$ top

top - 23:57:43 up  2:54,  1 user,  load average: 0.00, 0.01, 0.05
Tasks: 189 total,   2 running, 187 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.7%us,  3.0%sy,  0.0%ni, 96.3%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1026080k total,   924508k used,   101572k free,    37000k buffers
Swap:  1046524k total,    21472k used,  1025052k free,   367996k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1525 home      20   0 1775m 100m  28m S  1.7 10.0   5:05.34 Photoshop.exe
  961 root      20   0 75972  51m 7952 R  1.0  5.1   2:23.42 Xorg
 1507 home      20   0  7644 4652  696 S  1.0  0.5   2:42.66 wineserver
 1564 home      20   0 75144  29m 9840 S  0.3  3.0   0:25.96 ubuntuone-syncd
 2999 home      20   0  127m  13m  10m S  0.3  1.4   0:01.36 gnome-terminal
 3077 home      20   0  2820 1188  864 R  0.3  0.1   0:00.76 top
    1 root      20   0  3200 1704 1260 S  0.0  0.2   0:00.98 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.95 ksoftirqd/0
```

Press 'q' on the keyboard to move out of the process display.

The terminology follows:

Field	Description	Example 1	Example 2
PID	The process ID of each task	1525	961
User	The username of task owner	Home	Root
PR	Priority Can be 20(highest) or -20(lowest)	20	20
NI	The nice value of a task	0	0
VIRT	Virtual memory used (kb)	1775	75972
RES	Physical memory used (kb)	100	51
SHR	Shared memory used (kb)	28	7952
S	Status There are five types: 'D' = uninterruptible sleep 'R' = running 'S' = sleeping 'T' = traced or stopped 'Z' = zombie	S	R
%CPU	% of CPU time	1.7	1.0
%MEM	Physical memory used	10	5.1
TIME+	Total CPU time	5:05.34	2:23.42
Command	Command name	Photoshop.exe	Xorg

PS

This command stands for 'Process Status'. It is similar to the "Task Manager" that pop-ups in a Windows Machine when we use Cntrl+Alt+Del. This command is similar to 'top' command but the information displayed is different.

To check all the processes running under a user, use the command -

```
ps ux
```

```
home@VirtualBox:~$ ps ux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
home      1114  0.0  0.8  46548  8512 ?        Ssl  Sep03   0:00 gnome-sess
home      1151  0.0  0.0   3856   140 ?        Ss   Sep03   0:00 /usr/bin/s
home      1154  0.0  0.0   3748   484 ?        S    Sep03   0:00 /usr/bin/d
home      1155  0.1  0.2   6656  3036 ?        Ss   Sep03   0:18 //bin/dbus
home      1157  0.0  0.2   9148  2368 ?        S    Sep03   0:00 /usr/lib/g
home      1162  0.0  0.2  31588  2296 ?        Ssl  Sep03   0:00 /usr/lib/g
home      1174  0.0  1.4 132472 14884 ?        Sl   Sep03   0:03 /usr/lib/g
```

You can also check the process status of a single process, use the syntax -

```
ps PID
```

```
guru99@VirtualBox:~$ ps 1268
  PID TTY      STAT   TIME COMMAND
 1268 ?        S<l    0:02 /usr/bin/pulseaudio --start --log-target=syslog
```

Kill

This command **terminates running processes** on a Linux machine.

To use these utilities you need to know the PID (process id) of the process you want to kill

Syntax -

```
kill PID
```

To find the PID of a process simply type

```
pidof Process name
```

Let us try it with an example.

```
home@VirtualBox:~$ pidof Photoshop.exe
1525
home@VirtualBox:~$ kill 1525
```

NICE

Linux can run a lot of processes at a time, which can slow down the speed of some high priority processes and result in poor performance.

To avoid this, you can tell your machine to prioritize processes as per your requirements.

This priority is called Niceness in Linux, and it has a value between -20 to 19. The lower the Niceness index, the higher would be a priority given to that task.

The default value of all the processes is 0.

To start a process with a niceness value other than the default value use the following syntax

```
nice -n 'Nice value' process name
```

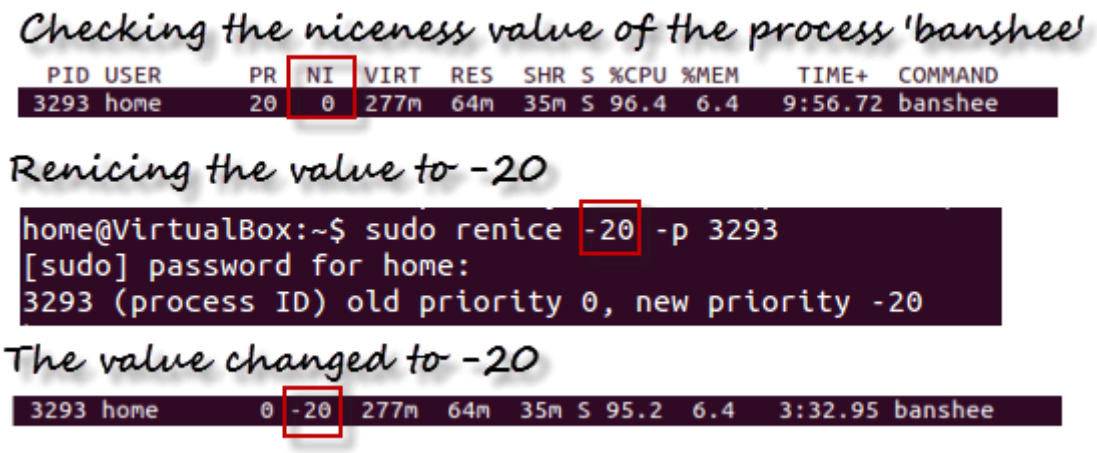
```
home@VirtualBox:~$ nice -n 19 banshee
```

If there is some process already running on the system, then you can 'Renice' its value using syntax.

```
renice 'nice value' -p 'PID'
```

To change Niceness, you can use the 'top' command to determine the PID (process id) and its Nice value. Later use the renice command to change the value.

Let us understand this by an example.



DF

This utility reports the free disk space(Hard Disk) on all the file systems.

```
guru99@guru99-VirtualBox:~$ df
Filesystem     1K-blocks    Used Available Use% Mounted on
/dev/sda1        7837756 2921376   4523216  40% /
udev              246488       4    246484   1% /dev
tmpfs             101512     752    100760   1% /run
none                5120       0      5120   0% /run/lock
none              253776      76    253700   1% /run/shm
```

If you want the above information in a readable format, then use the command

'df -h'

```
guru99@guru99-VirtualBox:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1       7.5G  2.8G  4.4G  40% /
udev            241M  4.0K  241M   1% /dev
tmpfs           100M  752K   99M   1% /run
none            5.0M     0  5.0M   0% /run/lock
none            248M   76K  248M   1% /run/shm
```

Free

This command shows the free and used memory (RAM) on the Linux system.

```
home@VirtualBox:~$ free
             total       used       free     shared    buffers     cached
Mem:       1026080     803604     222476          0      36312     343376
-/+ buffers/cache:     423916     602164
Swap:      1046524      35832    1010692
```

You can use the arguments

free -m to display output in MB

free -g to display output in GB

Summary:

- Any running program or a command given to a Linux system is called a process
- A process could run in foreground or background
- The priority index of a process is called Nice in Linux. Its default value is 0, and it can vary between 20 to -19
- The lower the Niceness index, the higher would be priority given to that task

Command	Description
bg	To send a process to the background
fg	To run a stopped process in the foreground
top	Details on all Active Processes
ps	Give the status of processes running for a user
ps PID	Gives the status of a particular process
pidof	Gives the Process ID (PID) of a process
kill PID	Kills a process
nice	Starts a process with a given priority

Command	Description
renice	Changes priority of an already running process
df	Gives free hard disk space on your system
free	Gives free RAM on your system