11.1 Introduction

In this chapter, you will learn how to run processes in the background or foreground, as well as how to make a process switch between the background and foreground. You will also be taught how to control processes by sending them signals using the `kill` command. In addition, different techniques for monitoring the resources that a process is using will be presented. Finally, you will see how to control the priority of processes to affect how much computing resources the processes will use.

11.2 Process Control

As mentioned in a previous chapter, running a command results in something called a *process*. In the Linux operating system, processes are executed with the privileges of the user who executes the command. This allows for processes to be limited to certain capabilities based upon the user identity. For example, typically a regular user cannot control another user's processes.

Although there are exceptions, generally the operating system will differentiate users based upon whether they are the administrator, also called the *root* user, or not. Non-root users are referred to as regular users. Users who have logged into the root account can control any user processes, including stopping any user process.

11.3 Process (ps) Command

The `ps` command can be used to list processes. Keep in mind that the `ps` command supports three styles of options:

- Traditional UNIX style short options that use a single hyphen in front of a character
- GNU style long options that use two hyphens in front of a word
- BSD style options that use no hyphens and single character options

```
sysadmin@localhost:~$ ps
  PID TTY          TIME CMD
   80 ?        00:00:00 bash
   94 ?        00:00:00 ps
```

The `ps` command will display the processes that are running in the current terminal by default. The output includes the following columns of information:

- `PID`: The process identifier, which is unique to the process. This information is useful to control the process by its ID number.
- `TTY`: The name of the terminal or pseudo-terminal where the process is running. This information is useful to distinguish between different processes that have the same name.
- `TIME`: The total amount of processor time used by the process. Typically, this information isn't used by regular users.
- `CMD`: The command that started the process.

When the `ps` command is run with a BSD style option then an additional column called `STAT` is displayed, which conveys the state of the processes. There are several states that a process can be in: `D` (Uninterruptible Sleep), `R` (Running), `S` (Interruptible Sleep), `T` (Stopped), and `Z` (Zombie).

When using a BSD style option, the `CMD` column is replaced with the `COMMAND` column, which shows not just the command, but also its options and arguments.

To see all of the current user's processes, use the BSD option `x`:

```
sysadmin@localhost:~$ ps x
  PID TTY      STAT   TIME COMMAND
   80 ?        S      0:00 -bash
   95 ?        R+     0:00 ps x
```

Instead of viewing just the processes running in the current terminal, users may want to view every process running on the system. With traditional (non-BSD) options, the `-e` option will display every process. Typically, the `-f` option is also used as it provides full details of the command, including options and arguments:

```
sysadmin@localhost:~$ ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 17:16 ?        00:00:00 /sbin??? /init
syslog        33     1  0 17:16 ?        00:00:00 /usr/sbin/rsyslogd
root          38     1  0 17:16 ?        00:00:00 /usr/sbin/cron
root          40     1  0 17:16 ?        00:00:00 /usr/sbin/sshd
bind          57     1  0 17:16 ?        00:00:00 /usr/sbin/named -u bind
root          70     1  0 17:16 ?        00:00:00 /bin/login -f
sysadmin      80    70  0 17:16 ?        00:00:00 -bash
sysadmin      96    80  0 17:26 ?        00:00:00 ps -ef
```

11.4 Foreground Processes

So far, the commands that have been presented in this course have been executing in what is known as the *foreground*. For simple commands that run quickly and then exit, using foreground execution is appropriate. A foreground process is one that prevents the user from using the shell until the process is complete.

When one process starts another, the first process is referred to as the *parent process* and the new process is called a *child process*. So, another way of thinking of a foreground process is that when run in the foreground, a child process doesn't allow any further commands to be executed in the parent process until the child process ends.

You do not need to add anything to a command in order to make that command execute in the foreground, as that is the default behavior.

11.5 Executing Multiple Commands

Before discussing background processes, consider how multiple commands can be executed on a single command line.

Normally, users only type one command per command line, but by using the `;` (semicolon character) as a delimiter between commands, a user can type multiple commands on one command line. Rarely is it really necessary to run two or more commands from one command line, but sometimes it can be useful.

When commands are separated by the semicolon, the command to the left of the semicolon executes; when it finishes, the command to the right of the semicolon executes. Another way of describing this is to say that the commands execute in sequence. For example:

```
sysadmin@localhost:~$ echo Hello;sleep 5; echo World
Hello
World
```

Note: The `sleep` command will pause for 5 seconds before continuing to the next command.

Recall that an alias is a feature that allows a user to create nicknames for commands or command lines. Having an alias that runs multiple commands can be very useful and to accomplish this, the user would use the `;` between each of the commands when creating the alias. For example, to create an alias called `welcome` which outputs the current user, the date, and the current directory listing, execute the following command:

```
sysadmin@localhost:~$ alias welcome="whoami;date;ls"
```

Now the alias `welcome` will execute `whoami`, `date`, and `ls` in series like so:

```
sysadmin@localhost:~$ welcome
sysadmin
Mon Sep  8 22:03:34 UTC 2014
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos  test
```

A real-life scenario involving multiple commands occurs when an administrator needs to take down and restart the network connection remotely. If the user typed the one command to bring down the network and then executed it, then the network connection would be terminated and the user wouldn't be able to bring the network back up again. Instead, the user could type the command to take down the network, followed by a semicolon, then the command to bring up the network. After pressing the **Enter** key, both commands would execute, one right after the other.

11.6 Background Processes

When a command may take some time to execute, it may be better to have that command execute in the "background". When executed in the background, a child process releases control back to the parent process (the shell, in this case) immediately, allowing the user to execute other commands. To have a command execute as a background process, add the **&** (ampersand character) after the command

To have multiple commands run in the background on one command line, place an ampersand after each command in that command line. In the next example, all three commands start nearly simultaneously and release control back to the shell, so the user does not have to wait for any of the commands to finish before executing another command (although the user might need to press **Enter** again to get a prompt):

```
sysadmin@localhost:~$ echo Hello&sleep 5&echo World&
[1] 103
[2] 104
[3] 105
Hello
sysadmin@localhost:~$ World

[1]   Done                    echo Hello
[2]-  Done                    sleep 5
[3]+  Done                    echo World
sysadmin@localhost:~$
```

There is an additional difference in executing commands in the background. After each command starts executing, it outputs [job number] followed by a space and then the *process identification number* (PID). These numbers are used for controlling the process, as you will see later.

After each background command finishes executing, it displays the job number, sometimes followed by a - or a + character, the word `Done` and then the command line itself. The meaning of the - and + will soon be explained.

While there are still background processes being run in the terminal, they can be displayed by using the `jobs` command. It is important to point out that the `jobs` command will only show background processes in the current terminal. If background processes are running in another terminal, they will not be shown by running the `jobs` command in the current terminal. The following is an example of using the `jobs` command:

```
sysadmin@localhost:~$ sleep 1000 &
[1] 106
sysadmin@localhost:~$ sleep 2000 &
```

```
[2] 107
sysadmin@localhost:~$ jobs
[1]-  Running                 sleep 1000 &
[2]+  Running                 sleep 2000 &
```

11.7 Moving Processes

If the `sleep` command from the previous page is run without the ampersand, the terminal would not be available for 1000 seconds:

```
sysadmin@localhost:~$ sleep 1000
```

To make the terminal available again, the administrator would have to use **CTRL+Z**:

```
^Z
[1]+  Stopped                 sleep 1000
```

Now the terminal is back, but the `sleep` command has been paused. To put the paused command in the background, execute the `bg` command. The `bg`command resumes jobs without bringing them to the foreground.

```
sysadmin@localhost:~$ bg
[1]+ sleep 1000 &
```

A command that has been paused or sent to the background can then be returned to the foreground using the `fg` command. To bring the `sleep`command back to the foreground, locking up the terminal again, use the `fg`command:

```
sysadmin@localhost:~$ fg
sleep 1000
```

Suppose we have two paused processes:

```
sysadmin@localhost:~$ sleep 1000
^Z
[1]+  Stopped                 sleep 1000
sysadmin@localhost:~$ sleep 2000
^Z
[2]+  Stopped                 sleep 2000
sysadmin@localhost:~$ jobs
[1]-  Stopped                 sleep 1000
[2]+  Stopped                 sleep 2000
```

Both `bg` and `fg` can take the *job number* as an argument to specify which process should be resumed. The following commands will resume `sleep 1000`in the background and resume `sleep 2000` in the foreground respectively:

```
sysadmin@localhost:~$ bg 1
[1]- sleep 1000 &
sysadmin@localhost:~$ fg 2
```

```
sleep 2000
```

It is also possible to use the name of the name of the command as an argument:

```
sysadmin@localhost:~$ sleep 1000
^Z
[1]+  Stopped                 sleep 1000
sysadmin@localhost:~$ bg sleep
[1]+ sleep 1000 &
```

Given multiple tasks, and only one terminal to use with them, the `fg` and `bg`commands provide an administrator with the ability to manually multi-task.

11.8 Sending a Signal

A signal is a message that is sent to a process to tell the process to take some sort of action, such as stop, restart, or pause. Signals are very useful in controlling the actions of a process.

Some signals can be sent to processes by simple keyboard combinations. For example, to have a foreground process paused, send a Terminal Stop signal by pressing **CTRL+Z**. A Terminal Stop pauses the program, but does not completely stop the program. To completely stop a foreground process, send the Interrupt signal by pressing **CTRL+C**.

There are many different signals, each of them have a symbolic name and a numeric identifier. For example, **CTRL+C** is assigned the symbolic name `SIGINT` and the numeric identifier of `2`.

To see a list of all of the signals available for your system, execute the `kill -l` command:

```
sysadmin@localhost:~$ 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
```

These signals can have unique meanings that are specific to a certain command (as the programmer who created the command can adjust the behavior of the program), but generally they allow for processes to be stopped and resumed, for processes to reconfigure themselves or to end a process. All the signals with a number greater than 31 are for controlling *real-time* processes, a topic which is beyond the scope of this course. Some of the more common signals are summarized in the following table:

Number	Full Name	Short Name	Purpose

Number	Full Name	Short Name	Purpose
1	SIGHUP	HUP	Hang up usually ends process
2	SIGINT	INT	Interrupt usually ends process
3	SIGQUIT	QUIT	Quit usually ends process
9	SIGKILL	KILL	Kill forcefully ends process
15	SIGTERM	TERM	Terminate usually ends process
18	SIGCONT	CONT	Continue resumes a stopped process
19	SIGSTOP	STOP	Stop forcefully stops a process
20	SIGTSTP	TSTP	Terminal Stop usually stops a process

There are several commands that will allow you to specify a signal to send to process; the `kill` command is the most typically used. The `kill` command accepts three different ways to specify the signal:

- The signal number used as an option: `-2`
- The short name of the signal used as an option: `-INT`
- The full name of the signal used as an option: `-SIGINT`

All three options shown above indicate the Interrupt signal.

If the user doesn't specify a signal with an option, then the `kill` command sends the Terminate `SIGTERM` signal.

When sending a signal, specify one or more processes to send the signal to. There are numerous techniques to specify the process or processes. The more common techniques include:

- Specifying the process identifier (PID)
- Using the `%` (percent sign) prefix to the job number
- Using the `-p` option along with the process identifier (PID)

For example, first imagine a scenario where a user is running some process in the background and that user wants to send a signal to the process. For this demonstration, the `sleep` command is run in the background:

```
sysadmin@localhost:~$ sleep 5000&
[1] 2901
```

A couple of items are noteworthy from the output of starting this process in the background. First, notice the job number in square brackets: `[1]`. Second, notice the process identifier (PID) which is `2901`. To send the Terminate signal, to this process, you can use any of the following commands:

- `kill 2901`
- `kill %1`

- `kill -p 2901`

As indicated earlier, the Terminate signal normally will end a process. Sometimes a process will *trap* the Terminate signal, so it may not end that process. Trapping occurs when a process behaves differently from the norm when it receives a signal; this can be the result of how the programmer created the code for the command.

A user could try to use other signals, like `SIGQUIT` or `SIGINT`, to try to end a process, but these signals can also be trapped. The only signal that will end a process and can't be trapped is `SIGKILL`. So, if other signals have failed to end a process, use the `SIGKILL` signal to force the process to end.

Users should not normally use the `SIGKILL` signal as the initial attempt to try to end a process because this forces the process to end immediately and will not allow the process the opportunity to "clean up" after itself. Processes often perform critical operations, such as deleting temporary files, when they exit naturally.

The following examples show how to send the "force kill" signal to a process:

- `kill -9 2901`
- `kill -KILL %1`
- `kill -SIGKILL -p 2901`

There are other commands that send processes signals, such as the `killall` and `pkill` commands, which are useful to stop many processes at once; typically the `kill` command is a good choice for sending signals to a single process.

Like the `kill` command, both the `killall` and `pkill` commands accept the three ways to specify a particular signal. Unlike the `kill` command, these other commands can be used to terminate all processes of a particular user with the `-u` option. For example, `killall -u bob` would stop all of the process owned by the user "bob".

The `killall` and `pkill` commands also accept the name of the process instead of a process or job ID. Just be careful as this may end up stopping more process than you had expected. An example of stopping a process using the process name:

```
sysadmin@localhost:~$ kill sleep
-bash: kill: sleep: arguments must be process or job IDs
sysadmin@localhost:~$ killall sleep
[1]+  Terminated              sleep 5000
```

11.9 HUP Signal

When a user logs off the system, all processes that are owned by that user are automatically sent the Hang-up signal `SIGHUP`. Typically, this signal causes those processes to end.

In some cases, a user may want to execute a command that won't automatically exit when it is sent a `HUP` signal. To have a process ignore a Hang-up signal, start the process with the `nohup` command.

For example, consider a scenario where a user has a script named `myjob.sh` that needs continue to run all night long. The user should start that script in the background of the terminal by executing:

```
sysadmin@localhost:~$ nohup myjob.sh &
```

After executing the script, the user could proceed to logout. The next time the user logs in, the output of the script, if any, would be contained in the `nohup.out` file in that user's home directory.

11.10 Process Priority

When a process runs, it needs to have access to the CPU to perform actions. Not all processes have the same access to the CPU. For example, system process typically have a higher *priority* when accessing the CPU.

The Linux kernel dynamically adjusts the priority of processes to try to make the operating system seem responsive to the user and efficient at performing tasks. A user can influence the priority that will be assigned to a process by setting a value of something called *niceness*.

The higher you set the niceness value, the lower the priority that will be assigned to a process. The default value of niceness for processes is zero; most user processes run at this nice value. Only a user with administrative (root) access can set negative niceness values or alter the niceness of an existing process to be a lower niceness value.

To set the initial niceness of a command, use the `nice` command as a prefix to the command to execute. For example, to execute the `cat /dev/zero > /dev/null` command at the lowest priority possible, execute the following command:

```
sysadmin@localhost:~$ nice -n 19 cat /dev/zero > /dev/null
```

If a user logs in as the root user, they could also execute the `cat /dev/zero > /dev/null` command with the highest priority possible by executing the following command:

```
root@localhost:~# nice -n -20 cat /dev/zero > /dev/null
```

To adjust the niceness of an existing process, use the `renice` command. This can be useful when the system becomes less responsive after running a CPU intensive command. A user could make the system more responsive again by making that process run "nicer".

To accomplish this, the user would need to discover the process identifier for the process by using the `ps` command and then use `renice` to adjust the priority back to normal. For example:

```
sysadmin@localhost:~$ su -
Password:
root@localhost:~# nice -n -20 cat /dev/zero > /dev/null &
[1] 121
root@localhost:~# ps
  PID TTY          TIME CMD
    1 ?        00:00:00 init
   70 ?        00:00:00 login
  108 ?        00:00:00 su
  109 ?        00:00:00 bash
  121 ?        00:00:04 cat
  122 ?        00:00:00 ps
root@localhost:~# renice -n 0 -p 121
121 (process ID) old priority -20, new priority 0
```

Note: The `su` command demonstrated above allows a regular user to temporarily become the root user. The user is required to enter the root account password, when prompted.

11.11 Monitoring Processes

While the `ps` command can display active processes, the `top` command provides the ability to monitor processes in real-time, as well as manage the processes. For example, the following graphic will demonstrate using the `top` command to monitor currently running processes, including three `cat` commands:

```
Tasks:  13 total,   4 running,   9 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us, 37.2 sy,  0.2 ni, 62.1 id,  0.0 wa,  0.1 hi,  0.0 si,  0.0 st
KiB Mem:  16438128 total, 13108516 used,  3329612 free,     4276 buffers
KiB Swap:        0 total,        0 used,        0 free.  9808716 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
  164 root      20   0    4364    696    616 R  87.4  0.1   1:23.32 cat
  165 root      30  10    4364    696    620 R   9.3  0.1   0:49.13 cat
  166 root      39  19    4364    772    696 R   1.7  0.1   0:41.75 cat
    1 root      20   0   17960   2972   2724 S   0.0  0.0   0:00.02 init
   33 syslog    20   0  255844   2728   2296 S   0.0  0.0   0:00.03 rsyslogd
   38 root      20   0   23656   2288   2076 S   0.0  0.0   0:00.00 cron
   40 root      20   0   61364   3124   2444 S   0.0  0.0   0:00.00 sshd
   57 bind      20   0  689640  29580   5328 S   0.0  0.2   0:00.13 named
   70 root      20   0   63132   2900   2452 S   0.0  0.0   0:00.00 login
   80 sysadmin  20   0   18176   3384   2896 S   0.0  0.0   0:00.04 bash
  151 root      20   0   46628   2708   2360 S   0.0  0.0   0:00.01 su
  152 root      20   0   18180   3388   2896 S   0.0  0.0   0:00.01 bash
  167 root      20   0   19860   2452   2124 R   0.0  0.0   0:00.00 top
```

Note that the `cat` command with the lowest niceness (`NI` column) is using the highest percentage CPU (`%CPU` column) while the `cat` command with the highest niceness is using the lowest percentage CPU.

The `top` command has numerous features; for example, it can be manipulated in an interactive manner. Pressing the `h` key while the `top` command is running will result in it displaying a "help" screen:

```
Help for Interactive Commands - procps-ng version 3.3.9
Window 1:Def: Cumulative mode Off.  System: Delay 3.0 secs; Secure mode Off.

  Z,B,E,e   Global: 'Z' colors; 'B' bold; 'E'/'e' summary/task memory scale
  l,t,m     Toggle Summary: 'l' load avg; 't' task/cpu stats; 'm' memory info
  0,1,2,3,I Toggle: '0' zeros; '1/2/3' cpus or numa node views; 'I' Irix mode
  f,F,X     Fields: 'f'/'F' add/remove/order/sort; 'X' increase fixed-width

  L,&,<,> . Locate: 'L'/'&' find/again; Move sort column: '<'/'>' left/right
  R,H,V,J . Toggle: 'R' Sort; 'H' Threads; 'V' Forest view; 'J' Num justify
  c,i,S,j . Toggle: 'c' Cmd name/line; 'i' Idle; 'S' Time; 'j' Str justify
  x,y     . Toggle highlights: 'x' sort field; 'y' running tasks
  z,b     . Toggle: 'z' color/mono; 'b' bold/reverse (only if 'x' or 'y')
  u,U,o,O . Filter by: 'u'/'U' effective/any user; 'o'/'O' other criteria
  n,#,^O  . Set: 'n'/'#' max tasks displayed; Show: Ctrl+'O' other filter(s)
  C,...   . Toggle scroll coordinates msg for: up,down,left,right,home,end
```

```
  k,r       Manipulate tasks: 'k' kill; 'r' renice
  d or s    Set update interval
  W,Y       Write configuration file 'W'; Inspect other output 'Y'
  q         Quit
          ( commands shown with '.' require a visible task display window )
Press 'h' or '?' for help with Windows,
Type 'q' or <Esc> to continue
```

The `k` and `r` keys are used to manage tasks or processes within the `top` program.

Pressing the `k` key will allow a user to "kill" or send a signal to a process. After pressing `k`, `top` will prompt for a PID and then for a signal to send to that process.

Pressing the `r` key will allow a user to "renice" a process by prompting for the PID and then the new niceness value.

11.12 Monitoring the System

There are also a couple of commands that can be used to monitor the overall state of the system: the `uptime` and `free` commands.

The `uptime` command shows the current time and the amount of time the system has been running, followed by the number of users who are currently logged in and the load averages during the past one, five and fifteen minute intervals.

The numbers reported for the load averages are based upon how many CPU cores are available. Think of each processor as having 100% resources (CPU time) available. One processor = 100%, four processors = 400%. The `uptime` command is reporting the amount of resources used, but it is dividing by 100. So 1.00 is actually 100% of the CPU time being used, 2.00 is 200% and so on.

If a system has only one CPU core, then a value of 1.00 indicates that the system was fully loaded with tasks. If the system has two CPU cores, then a value of 1.00 would indicate a 50% load (1.00/2.00). An `uptime` reading of 1.00 (or 100%) usage on a 4 core CPU would imply that 1.00/4.00 (or 1/4 (or 25%)) of the CPU's total computational resources are being used.

To get a good idea of how busy a system is, use the `uptime` command:

```
sysadmin@localhost:~$ uptime
18:24:58 up 5 days, 10:43,  1 user,  load average: 0.08, 0.03, 0.05
```

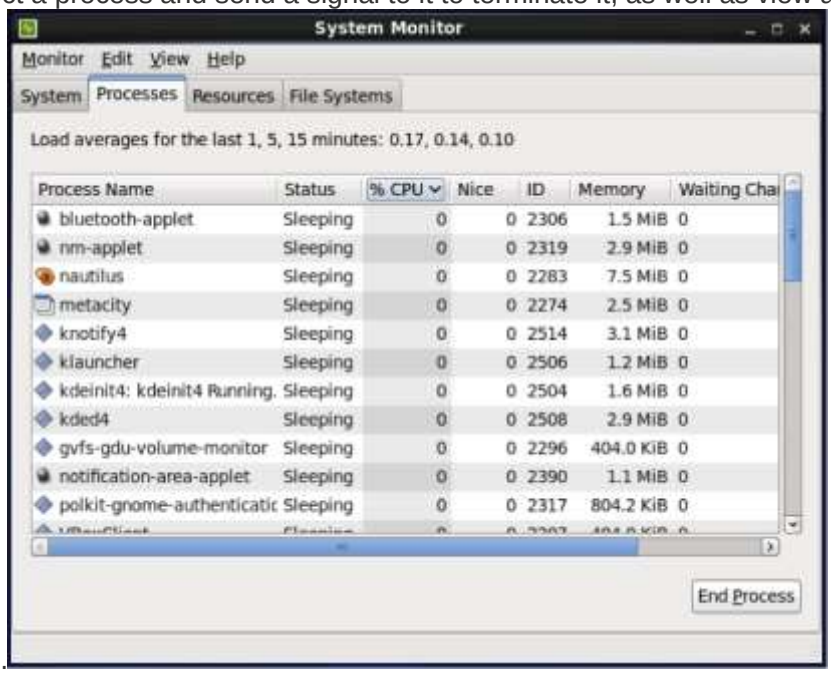
To get an idea of how your system is using memory, the `free` command is helpful. This command displays not only the values for the random access memory (RAM) but also for swap, which is space on the hard disk that is used as memory for idle processes when the random access memory starts to become scarce.

The `free` command reports the total amount of memory, as well as how much is currently used and how much is free to be used. The output also breaks down the use of memory for buffers and caches:

```
sysadmin@localhost:~$ free
             total       used       free     shared    buffers     cached
Mem:      16438128   13106024    3332104       3200       4276    9808896
-/+ buffers/cache:    3292852   13145276
Swap:            0          0          0
```

If logged into the Gnome desktop environment, the user can use the `gnome-system-monitor`. This tool (pictured below) is analogous to the Task Manager in Microsoft Windows. It provides four tabs to view information about the System, Processes, Resources and File Systems.

This graphical tool allows users to select a process and send a signal to it to terminate it, as well as view all current processes updated in real-time.



Note: The K desktop environment (KDE) has a similar command called `ksysguard`.