**Chapter 16: Advanced Bash Scripting**

**Learning Objectives**



By the end of this chapter, you should be able to:

* Manipulate strings to perform actions such as comparison and sorting.
* Use Boolean expressions when working with multiple data types including strings or numbers as well as files.
* Use case statements to handle command line options.
* Use looping constructs to execute one or more lines of code repetitively.
* Debug scripts using set -x and set +x.
* Create temporary files and directories.

Create and use random numbers.

**String Manipulation**

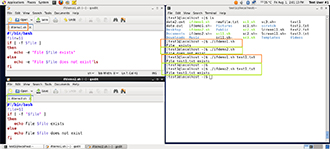
Let’s go deeper and find out how to work with strings in scripts.

A **string variable** contains a sequence of text characters. It can include letters, numbers, symbols and punctuation marks. Some examples: abcde, 123, abcde 123, abcde-123, &acbde=%123

String **operators** include those that do comparison, sorting, and finding the length. The following table demonstrates the use of some basic string operators.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| [ string1 > string2 ] | Compares the sorting order of string1 and string2. |
| [ string1 == string2 ] | Compares the characters in string1 with the characters in string2. |
| myLen1=${#mystring1} | Saves the length of string1 in the variable myLen1. |

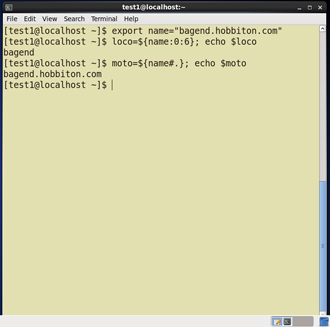
**Example of String Manipulation**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen04.jpg)

In the first part of the example, we compare the first string with the second string and display an appropriate message using the if statement.

In the second part of the example, we compare the string values returned by a command with the string value stored in an environmental variable.

**Parts of a String**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen05a.jpg)

At times, you may not need to compare or use an entire string. To extract the first character of a string we can specify:

${string:0:1} Here 0 is the offset in the string (i.e., which character to begin from) where the extraction needs to start and 1 is the number of characters to be extracted.

To extract all characters in a string after a dot (.), use the following expression: ${string#\*.}

**Boolean Expressions**

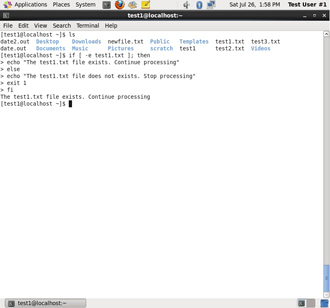
**Boolean** expressions evaluate to either **TRUE** or **FALSE**, and results are obtained using the various Boolean operators listed in the table.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operation** | **Meaning** |
| **&&** | **AND** | The action will be performed only if both the conditions evaluate to true. |
| **||** | **OR** | The action will be performed if any one of the conditions evaluate to true. |
| **!** | **NOT** | The action will be performed only if the condition evaluates to false. |

Note that if you have multiple conditions strung together with the && operator processing stops as soon as a condition evaluates to false. For example if you have A && B && C and A is true but B is false, C will never be executed.

Likewise if you are using the || operator, processing stops as soon as anything is true. For example if you have A || B || C and A is false and B is true, you will also never execute C.

**Tests in Boolean Expressions**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen10a.jpg)

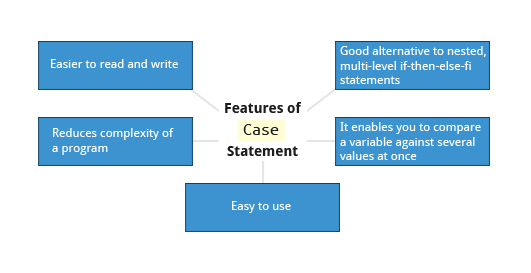
Boolean expressions return either **TRUE** or **FALSE**. We can use such expressions when working with multiple data types including strings or numbers as well as with files. For example, to check if a file exists, use the following conditional test:

[ -e <filename> ]  Similarly, to check if the value of number1 is greater than the value of number2, use the following conditional test:

 [ $number1 -gt $number2 ]  The operator -gt returns **TRUE** if number1 is greater than number2.

Click the image to view an enlarged version.

**The case Statement**



The case statement is used in scenarios where the actual value of a variable can lead to different execution paths. case statements are often used to handle command-line options.

Below are some of the advantages of using the case statement:

* It is easier to read and write.
* It is a good alternative to nested, multi-level if-then-else-fi code blocks.
* It enables you to compare a variable against several values at once.

It reduces the complexity of a program.

**Structure of the case Statement**

Here is the basic structure of the case statement:

case expression in

pattern1) execute commands;;

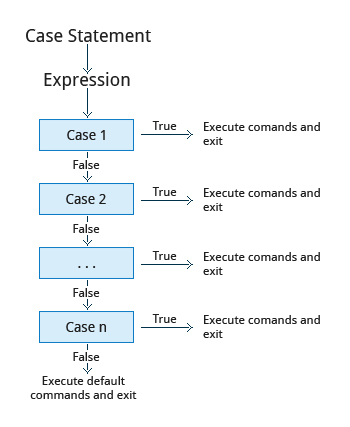
pattern2) execute commands;;

pattern3) execute commands;;

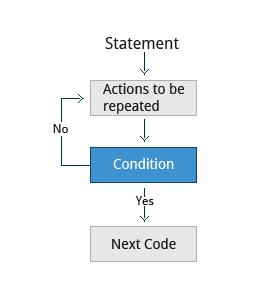
pattern4) execute commands;;

\* )       execute some default commands or nothing ;;

esac



**Looping Constructs**



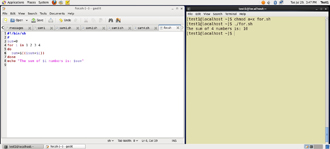
By using **looping constructs**, you can execute one or more lines of code repetitively. Usually you do this until a conditional test returns either true or false as is required.

Three type of loops are often used in most programming languages:

* for
* while
* until

All these loops are easily used for repeating a set of statements until the exit condition is true.

**The 'for' Loop**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen26a.jpg)

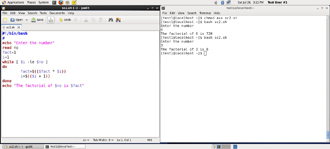
The for loop operates on each element of a list of items. The syntax for the for loop is:

for *variable-name* in *list* do     execute one iteration for each item in the             *list* until the *list* is finished done

In this case, *variable-name* and *list* are substituted by you as appropriate (see examples). As with other looping constructs, the statements that are repeated should be enclosed by do and done.

The screenshots here show an example of the for loop to print the sum of numbers 1 to 4.

**The while Loop**

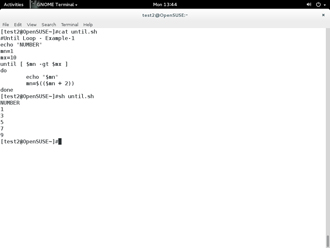
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen24a.jpg)

The while loop repeats a set of statements as long as the control command returns true. The syntax is:  while condition is true do     Commands for execution     ---- done

The set of commands that need to be repeated should be enclosed between do and done. You can use any command or operator as the condition.  Often it is enclosed within square brackets ([]).

The screenshots here show an example of the **while** loop that calculates the factorial of a number.

**The until loop**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen25a.jpg)

The until loop repeats a set of statements as long as the control command is false. Thus it is essentially the opposite of the while loop. The syntax is:

until condition is false do     Commands for execution     ---- done

Similar to the while loop, the set of commands that need to be repeated should be enclosed between do and done. You can use any command or operator as the condition.

The screenshot here shows example of the until loop that displays odd numbers between 1 and 10.

**Introduction to Script Debugging**



While working with scripts and commands, you may run into errors. These may be due to an error in the script, such as incorrect syntax, or other ingredients such as a missing file or insufficient permission to do an operation. These errors may be reported with a specific error code, but often just yield incorrect or confusing output. So how do you go about identifying and fixing an error?

**Debugging** helps you troubleshoot and resolve such errors, and is one of the most important tasks a system administrator performs.

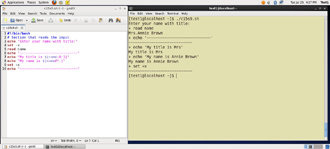
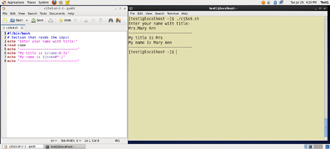
**More About Script Debugging**

Before fixing an error (or bug), it is vital to know its source.

In **bash** shell scripting, you can run a script in **debug mode** by doing bash –x ./script\_file. Debug mode helps identify the error because:

* It traces and prefixes each command with the + character.
* It displays each command before executing it.
* It can debug only selected parts of a script (if desired) with: set -x    # turns on debugging ... set +x    # turns off debugging

The screenshots shown here demonstrate a scriptfile called sc2 (left image) and the results from running it in debug mode (right image).

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen31B.jpg) [](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen30A.jpg)

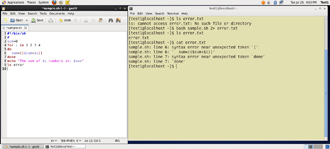
Click the image to view an enlarged version.

**Redirecting Errors to File and Screen**

In UNIX/Linux, all programs that run are given three open file streams when they are started as listed in the table:

|  |  |  |
| --- | --- | --- |
| **File stream** | **Description** | **File Descriptor** |
| **stdin** | Standard Input, by default the keyboard/terminal for programs run from the command line | 0 |
| **stdout** | Standard output, by default the screen for programs run from the command line | 1 |
| **stderr** | Standard error, where output error messages are shown or saved | 2 |

Using redirection we can save the **stdout** and **stderr** output streams to one file or two separate files for later analysis after a program or command is executed

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen32.jpg)

On the left screen is a buggy shell script. On the right screen the buggy script is executed and the errors are redirected to the file "error.txt". Using "cat" to display the contents of "error.txt" shows the errors of executing the buggy shell script (presumably for further debugging).

Click the image to view an enlarged version.

**Creating Temporary Files and Directories**

Consider a situation where you want to retrieve 100 records from a file with 10,000 records. You will need a place to store the extracted information, perhaps in a **temporary file**, while you do further processing on it.

Temporary files (and directories) are meant to store data for a short time. Usually one arranges it so that these files disappear when the program using them terminates. While you can also use **touch**to create a temporary file, this may make it easy for hackers to gain access to your data.

The best practice is to create random and unpredictable filenames for temporary storage. One way to do this is with the **mktemp**utility as in these examples:

The XXXXXXXX is replaced by the **mktemp**utility with random characters to ensure the name of the temporary file cannot be easily predicted and is only known within your program.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| TEMP=$(mktemp /tmp/tempfile.XXXXXXXX) | To create a temporary file |
| TEMPDIR=$(mktemp -d /tmp/tempdir.XXXXXXXX) | To create a temporary directory |

**Example of Creating a Temporary File and Directory**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen36.jpg)

First, the danger: If someone creates a symbolic link from a known temporary file used by root to the /etc/passwd file, like this:

$ ln -s /etc/passwd /tmp/tempfile There could be a big problem if a script run by root has a line in like this:

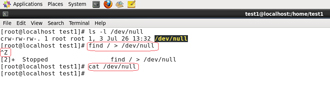
echo $VAR > /tmp/tempfile

The password file will be overwritten by the temporary file contents.

To prevent such a situation make sure you randomize your temporary filenames by replacing the above line with the following lines:

TEMP=$(mktemp /tmp/tempfile.XXXXXXXX) echo $VAR > $TEMP

**Discarding Output with /dev/null**

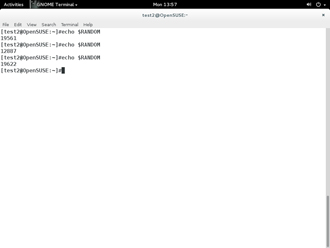
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen37.jpg)

Certain commands like **find** will produce voluminous amounts of output which can overwhelm the console. To avoid this, we can redirect the large output to a special file (a device node) called **/dev/null**. This file is also called the **bit bucket** or **black hole**.

It discards all data that gets written to it and never returns a failure on write operations. Using the proper redirection operators, it can make the output disappear from commands that would normally generate output to **stdout** and/or **stderr**:

$ find / > /dev/null In the above command, the entire standard output stream is ignored, but any errors will still appear on the console.

**Random Numbers and Data**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen38a.jpg)

It is often useful to generate random numbers and other random data when performing tasks such as:

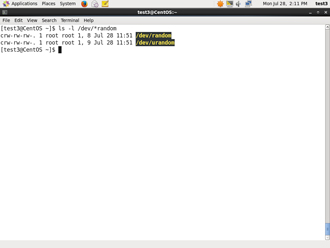
* Performing security-related tasks.
* Reinitializing storage devices.
* Erasing and/or obscuring existing data.
* Generating meaningless data to be used for tests.

Such random numbers can be generated by using the $RANDOM  environment variable, which is derived from the Linux kernel’s built-in random number generator, or by the **OpenSSL** library function, which uses the FIPS140 algorithm to generate random numbers for encryption

To read more about FIPS140, see <http://en.wikipedia.org/wiki/FIPS_140-2>

The example shows you how to easily use the environmental variable method to generate random numbers.

**How the Kernel Generates Random Numbers**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch15_screen39a.jpg)

Some servers have hardware random number generators that take as input different types of noise signals, such as thermal noise and photoelectric effect. A **transducer** converts this noise into an electric signal, which is again converted into a digital number by an **A-D converter**.  This number is considered random. However, most common computers do not contain such specialized hardware and instead rely on events created during booting to create the raw data needed.

Regardless of which of these two sources is used, the system maintains a so-called **entropy pool** of these digital numbers/random bits. Random numbers are created from this entropy pool.

The Linux kernel offers the **/dev/random** and **/dev/urandom** device nodes which draw on the entropy pool to provide random numbers which are drawn from the estimated number of bits of noise in the entropy pool.

**/dev/random** is used where very high quality randomness is required, such as one-time pad or key generation, but it is relatively slow to provide vaules.   **/dev/urandom** is faster and suitable (good enough) for most cryptographic purposes.

Furthermore, when the entropy pool is empty, **/dev/random** is blocked and does not generate any number until additional environmental noise (network traffic, mouse movement, etc.) is gathered whereas **/dev/urandom** reuses the internal pool to produce more pseudo-random bits.

**Summary (1 of 2)**



You have completed this chapter. Let’s summarize the key concepts covered:

* You can manipulate strings to perform actions such as comparison, sorting, and finding length.
* You can use Boolean expressions when working with multiple data types including strings or numbers as well as files.
* The output of a Boolean expression is either true or false.

Operators used in Boolean expressions include the && (AND), ||(OR), and!(NOT) operators.

**Summary (2 of 2)**



* We looked at the advantages of using case statement in scenarios where the value of a variable can lead to different execution paths.
* Script debugging methods help troubleshoot and resolve errors.
* The standard and error outputs from a script or shell commands can easily be redirected into the same file or separate files to aid in debugging and saving results
* Linux allows you to create temporary files and directories, which store data for a short duration, both saving space and increasing security.
* Linux provides several different ways of generating random numbers, which are widely used.

**Chapter 17: Processes**

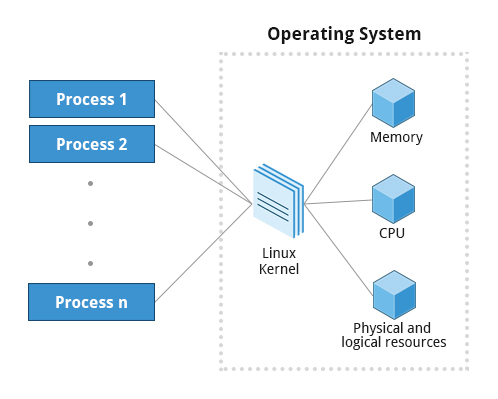
**Learning Objectives**



By the end of this chapter, you should be able to:

* Describe what a process is and distinguish between types of processes.
* Enumerate process attributes.
* Manage processes using **ps** and **top**.
* Understand the use of load averages and other process metrics.
* Manipulate processes by putting them in **background** and restoring them to **foreground**.
* Use **at**, **cron**, and **sleep** to schedule processes in the future or pause them.

**What Is a Process?**



**A process** is simply an instance of one or more related **tasks (threads)**executing on your computer. It is not the same as a **program** or a **command**; a single program may actually start several processes simultaneously. Some processes are independent of each other and others are related. A failure of one process may or may not affect the others running on the system.

Processes use many system resources, such as memory, CPU (central processing unit) cycles, and peripheral devices such as printers and displays. The operating system (especially the kernel) is responsible for allocating a proper share of these resources to each process and ensuring overall optimum utilization.

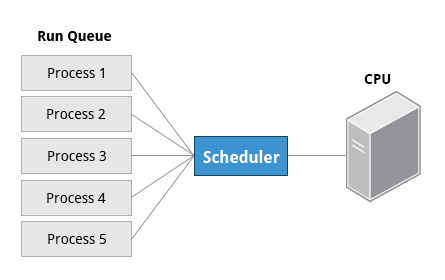
**Process Types**

A terminal window (one kind of command shell), is a process that runs as long as needed. It allows users to execute programs and access resources in an interactive environment. You can also run programs in the **background**, which means they become **detached** from the shell.

|  |  |  |
| --- | --- | --- |
| **Process Type** | **Description** | **Example** |
| Interactive Processes | Need to be started by a user, either at a command line or through a graphical interface such as an icon or a menu selection. | **bash, firefox, top** |
| Batch Processes | Automatic processes which are scheduled from and then disconnected from the terminal. These tasks are queued and work on a **FIFO** (First In, First Out) basis. | **updatedb** |
| Daemons | Server processes that run continuously. Many are launched during system startup and then wait for a user or system request indicating that their service is required. | **httpd, xinetd, sshd** |
| Threads | Lightweight processes. These are **tasks** that run under the umbrella of a main process, sharing memory and other resources, but are scheduled and run by the system on an individual basis. An individual thread can end without terminating the whole process and a process can create new threads at any time. Many non-trivial programs are multi-threaded. | **gnome-terminal, firefox** |
| Kernel Threads | Kernel tasks that users neither start nor terminate and have little control over. These may perform actions like moving a thread from one CPU to another, or making sure input/output operations to disk are completed. | **kswapd0, migration, ksoftirqd** |

Processes can be of different types according to the task being performed. Here are some different process types along with their descriptions and examples.

**Process Scheduling and States**



When a process is in a so-called **running** state, it means it is either currently executing instructions on a CPU, or is waiting for a share (or **time slice)** so it can run. A critical kernel routine called the **scheduler** constantly shifts processes in and out of the CPU, sharing time according to relative priority, how much time is needed and how much has already been granted to a task. All processes in this state reside on what is called a **run queue** and on a computer with multiple CPUs, or cores, there is a run queue on each.

However, sometimes processes go into what is called a **sleep** state, generally when they are waiting for something to happen before they can resume, perhaps for the user to type something. In this condition a process is sitting in a **wait** queue.

There are some other less frequent process states, especially when a process is terminating. Sometimes a child process completes but its parent process has not asked about its state. Amusingly such a process is said to be in a **zombie** state; it is not really alive but still shows up in the system's list of processes.

**Process and Thread IDs**

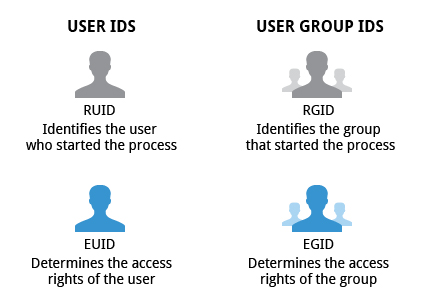
At any given time there are always multiple processes being executed. The operating system keeps track of them by assigning each a unique **process ID** (**PID**) number. The PID is used to track process state, cpu usage, memory use, precisely where resources are located in memory, and other characteristics.

New PIDs are usually assigned in ascending order as processes are born. Thus PID 1 denotes the **init** process (initialization process), and succeeding processes are gradually assigned higher numbers.

The table explains the PID types and their descriptions:

|  |  |
| --- | --- |
| **ID Type** | **Description** |
| Process ID (PID) | Unique Process ID number |
| Parent Process ID (PPID) | Process (Parent) that started this process |
| Thread ID (TID) | Thread ID number. This is the same as the PID for single-threaded processes. For a multi-threaded process, each thread shares the same PID but has a unique TID. |

**User and Group IDs**



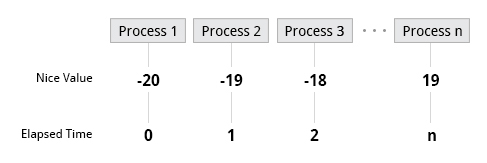
Many users can access a system simultaneously, and each user can run multiple processes. The operating system identifies the user who starts the process by the Real User ID (**RUID**) assigned to the user.

The user who determines the access rights for the users is identified by the Effective UID (**EUID**). The EUID may or may not be the same as the RUID.

Users can be categorized into various groups. Each group is identified by the Real Group ID, or **RGID**. The access rights of the group are determined by the Effective Group ID, or **EGID**. Each user can be a member of one or more groups.

Most of the time we ignore these details and just talk about the User ID (**UID**).

**More About Priorities**

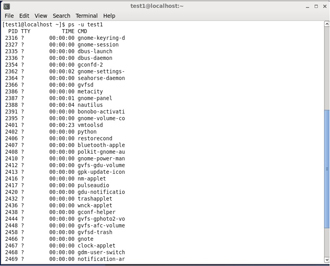


At any given time, many processes are running (i.e., in the run queue) on the system. However, a CPU can actually accommodate only one task at a time, just like a car can have only one driver at a time. Some processes are more important than others so Linux allows you to set and manipulate process **priority**. Higher priority processes are granted more time on the CPU.

The priority for a process can be set by specifying a **nice value**, or **niceness**, for the process. The lower the nice value, the higher the priority. Low values are assigned to important processes, while high values are assigned to processes that can wait longer. A process with a high nice value simply allows other processes to be executed first. In Linux, a nice value of -20 represents the highest priority and 19 represents the lowest. (This does sound kind of backwards, but this convention, the nicer the process, the lower the priority, goes back to the earliest days of UNIX.)

You can also assign a so-called **real-time priority** to time-sensitive tasks, such as controlling machines through a computer or collecting incoming data. This is just a very high priority and is not to be confused with what is called **hard real time** which is conceptually different, and has more to do with making sure a job gets completed within a very well-defined time window.

**The ps Command (System V Style)**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen14.jpg)

**ps**provides information about currently running processes, keyed by **PID**. If you want a repetitive update of this status, you can use **top** or commonly installed variants such as **htop**or **atop** from the command line, or invoke your distribution's graphical system monitor application.

**ps**has many options for specifying exactly which tasks to examine, what information to display about them, and precisely what output format should be used.

Without options **ps**will display all processes running under the current shell. You can use the -u option to display information of processes for a specified username. The command ps -ef displays all the processes in the system in full detail. The command ps -eLf goes one step further and displays one line of information for every **thread** (remember, a process can contain multiple threads).

**The ps Command (BSD Style)**

**ps**has another style of option specification which stems from the **BSD** variety of UNIX, where options are specified without preceding dashes. For example, the command ps aux displays all processes of all users. The command ps axo allows you to specify which attributes you want to view.

The following table shows sample output of **ps** with the aux and axo qualifiers.

|  |  |
| --- | --- |
| **Command** | **Output** |
| ps aux | USER PID %CPU %MEM VSZ   RSS  TTY STAT START TIME COMMAND root 1   0.0  0.0  19356 1292 ?   Ss   Feb27 0:08 /sbin/init root 2   0.0  0.0  0     0    ?   S    Feb27 0:00 [kthreadd] root 3   0.0  0.0  0     0    ?   S    Feb27 0:27 [migration/0]   . . . |
| ps axo stat, priority, pid, pcpu, comm | STAT PRI PID %CPU COMMAND Ss   20  1   0.0  init S    20  2   0.0  kthreadd S   -100 3   0.0  migration/0   . . . |

**The Process Tree**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen18.jpg)

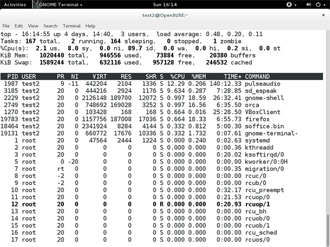
At some point one of your applications may stop working properly. How might you terminate it?

**pstree**displays the processes running on the system in the form of a **tree diagram**showing the relationship between a process and its parent process and any other processes that it created. Repeated entries of a process are not displayed, and threads are displayed in curly braces.

To terminate a process you can type kill -SIGKILL <pid> or kill -9 <pid>. Note however, you can only **kill** your own processes: those belonging to another user are off limits unless you are root.

Click the image to view an enlarged version.

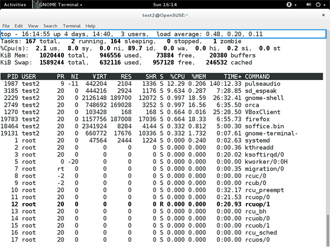
**top**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen19.jpg)

While a static view of what the system is doing is useful, monitoring the system performance live over time is also valuable. One option would be to run **ps**at regular intervals, say, every two minutes. A better alternative is to use **top** to get constant real-time updates (every two seconds by default) until you exit by typing q. **top** clearly highlights which processes are consuming the most CPU cycles and memory (using appropriate commands from within **top**.)

Click the image to view an enlarged version.

**First Line of the top Output**

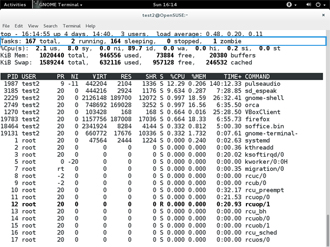
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen20.jpg)

The first line of the **top** output displays a quick summary of what is happening in the system including:

* How long the system has been up
* How many users are logged on
* What is the load average

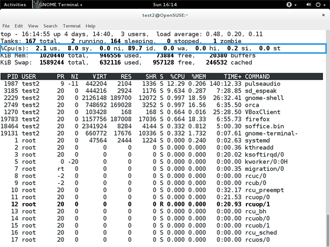
The **load average** determines how busy the system is. A load average of 1.00 per CPU indicates a fully subscribed, but not overloaded, system. If the load average goes above this value, it indicates that processes are competing for CPU time. If the load average is very high, it might indicate that the system is having a problem, such as a runaway process (a process in a non-responding state).

**Second Line of the top Output**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen21.jpg)

The second line of the **top** output displays the total number of processes, the number of running, sleeping, stopped and zombie processes. Comparing the number of running processes with the load average helps determine if the system has reached its capacity or perhaps a particular user is running too many processes. The stopped processes should be examined to see if everything is running correctly.

**hird Line of the top Output**

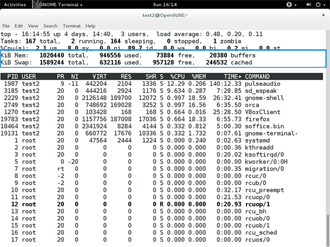
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen22.jpg)

The third line of the **top**output indicates how the CPU time is being divided between the users (**us**) and the kernel (**sy**) by displaying the percentage of CPU time used for each.

The percentage of user jobs running at a lower priority (niceness - **ni**) is then listed. Idle mode (**id**) should be low if the load average is high, and vice versa. The percentage of jobs waiting (**wa**) for I/O is listed. Interrupts include the percentage of hardware (**hi**) vs. software interrupts (**si**). Steal time (**st**) is generally used with virtual machines, which has some of its idle CPU time taken for other uses.

Click the image to view an enlarged version.

**Fourth and Fifth Lines of the top Output**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen23.jpg)

The fourth and fifth lines of the **top**output indicate memory usage, which is divided in two categories:

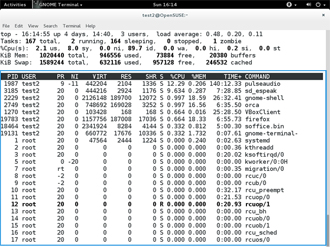
* Physical memory (RAM) – displayed on line 4.
* Swap space – displayed on line 5.

Both categories display total memory, used memory, and free space.

You need to monitor memory usage very carefully to ensure good system performance. Once the physical memory is exhausted, the system starts using **swap** space (temporary storage space on the hard drive) as an extended memory pool, and since accessing disk is much slower than accessing memory, this will negatively affect system performance.

If the system starts using swap often, you can add more swap space. However, adding more physical memory should also be considered.

**Process List of the top Output**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen24.jpg)

Each line in the process list of the **top**output displays information about a process. By default, processes are ordered by highest CPU usage. The following information about each process is displayed:

* Process Identification Number (PID)
* Process owner (USER)
* Priority (PR) and nice values (NI)
* Virtual (VIRT), physical (RES), and shared memory (SHR)
* Status (S)
* Percentage of CPU (%CPU) and memory (%MEM) used
* Execution time (TIME+)
* Command (COMMAND)

Click the image to view an enlarged version.

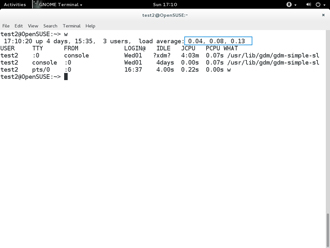
**Interactive Keys with top**

Besides reporting information, **top** can be utilized interactively for monitoring and controlling processes. While **top** is running in a terminal window you can enter single-letter commands to change its behaviour. For example, you can view the top-ranked processes based on CPU or memory usage. If needed, you can alter the priorities of running processes or you can stop/kill a process.

The table lists what happens when pressing various keys when running **top**:

|  |  |
| --- | --- |
| **Command** | **Output** |
| t | Display or hide summary information (rows 2 and 3) |
| m | Display or hide memory information (rows 4 and 5) |
| A | Sort the process list by top resource consumers |
| r | Renice (change the priority of) a specific processes |
| k | Kill a specific process |
| f | Enter the top configuration screen |
| o | Interactively select a new sort order in the process list |

**Load Averages**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen33.jpg)

**Load average** is the average of the **load number** for a givenperiod of time. It takes into account processes that are:

* Actively running on a CPU.
* Considered runnable, but waiting for a CPU to become available.
* Sleeping: i.e., waiting for some kind of resource (typically, I/O) to become available.

The load average can be obtained by running **w**, **top** or **uptime**.

Click the image to view an enlarged version.

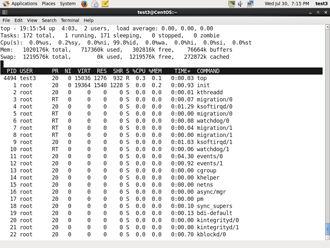
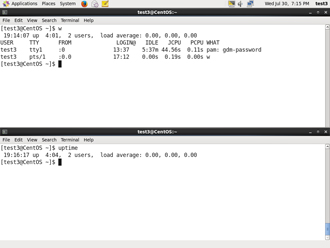
**Interpreting Load Averages**

The load average is displayed using three different sets of numbers, as shown in the following example:

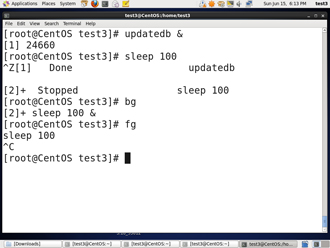
The last piece of information is the average load of the system. Assuming our system is a single-CPU system, the 0.25 means that for the past minute, on average, the system has been 25% utilized. 0.12 in the next position means that over the past 5 minutes, on average, the system has been 12% utilized; and 0.15 in the final position means that over the past 15 minutes, on average, the system has been 15% utilized. If we saw a value of 1.00 in the second position, that would imply that the single-CPU system was 100% utilized, on average, over the past 5 minutes; this is good if we want to fully use a system. A value over 1.00 for a single-CPU system implies that the system was over-utilized: there were more processes needing CPU than CPU was available.

If we had more than one CPU, say a quad-CPU system, we would divide the load average numbers by the number of CPUs. In this case, for example, seeing a 1 minute load average of 4.00 implies that the system as a whole was 100% (4.00/4) utilized during the last minute.

Short term increases are usually not a problem. A high peak you see is likely a burst of activity, not a new level. For example, at start up, many processes start and then activity settles down. If a high peak is seen in the 5 and 15 minute load averages, it would may be cause for concern.

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen34a.jpg) [](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen34b.jpg)

**Background and Foreground Processes**

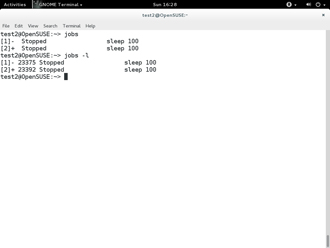
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen35.jpg)

Linux supports **background** and **foreground** job processing. (A job in this context is just a command launched from a terminal window.) **Foreground** jobs run directly from the shell, and when one foreground job is running, other jobs need to wait for shell access (at least in that terminal window if using the GUI) until it is completed. This is fine when jobs complete quickly. But this can have an adverse effect if the current job is going to take a long time (even several hours) to complete.

In such cases, you can run the job in the **background** and free the shell for other tasks. The background job will be executed at lower priority, which, in turn, will allow smooth execution of the interactive tasks, and you can type other commands in the terminal window while the background job is running. By default all jobs are executed in the foreground. You can put a job in the background by suffixing & to the command, for example: updatedb &

You can either use **CTRL-Z** to suspend a foreground job or **CTRL-C** to terminate a foreground job and can always use the **bg** and **fg**commands to run a process in the background and foreground, respectively.

**Managing Jobs**

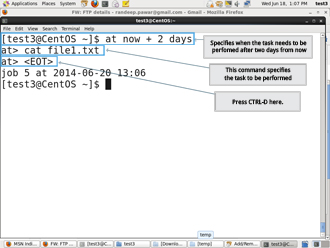
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen36.jpg)

The **jobs** utility displays all jobs running in background. The display shows the job ID, state, and command name, as shown here.

jobs -l provides a the same information as jobs including the PID of the background jobs.

The background jobs are connected to the terminal window, so if you log off, the **jobs** utility will not show the ones started from that window.

**Scheduling Future Processes using at**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen41.jpg)

Suppose you need to perform a task on a specific day sometime in the future. However, you know you will be away from the machine on that day. How will you perform the task? You can use the **at** utility program to execute any non-interactive command at a specified time, as illustrated in the diagram:

Click the image to view an enlarged version.

**cron**

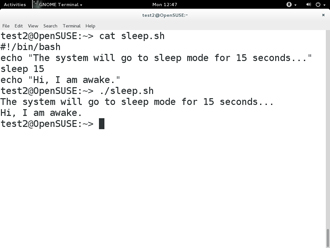
**cron** is a time-based scheduling utility program. It can launch routine background jobs at specific times and/or days on an on-going basis. **cron** is driven by a configuration file called /etc/crontab (**cron** table) which contains the various shell commands that need to be run at the properly scheduled times. There are both system-wide crontab files and individual user-based ones. Each line of a crontab file represents a job, and is composed of a so-called CRON expression, followed by a shell command to execute.

The crontab -e command will open the crontab editor to edit existing jobs or to create new jobs. Each line of the crontab file will contain 6 fields:

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | **Values** |
| MIN | Minutes | 0 to 59 |
| HOUR | Hour field | 0 to 23 |
| DOM | Day of Month | 1-31 |
| MON | Month field | 1-12 |
| DOW | Day Of Week | 0-6 (0 = Sunday) |
| CMD | Command | Any command to be executed |

Examples: 1. The entry "\* \* \* \* \* /usr/local/bin/execute/this/script.sh" will schedule a job to execute 'script.sh' every minute of every hour of every day of the month, and every month and every day in the week.  2. The entry "30 08 10 06 \* /home/sysadmin/full-backup" will schedule a full-backup at 8.30am, 10-June irrespective of the day of the week.

**sleep**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch16_screen43.jpg)

Sometimes a command or job must be delayed or suspended. Suppose, for example, an application has read and processed the contents of a data file and then needs to save a report on a backup system. If the backup system is currently busy or not available, the application can be made to**sleep** (wait) until it can complete its work. Such a delay might be to mount the backup device and prepare it for writing.

**sleep**suspends execution for at least the specified period of time, which can be given as the number of seconds (the default), minutes, hours or days. After that time has passed (or an interrupting signal has been received) execution will resume.

Syntax: sleep NUMBER[SUFFIX]...         where SUFFIX may be:                 1.   s for seconds (the default)                 2.   m for minutes                 3.   h for hours                 4.   d for days

**sleep** and **at** are quite different; **sleep** delays execution for a specific period while **at** starts execution at a later time.

**Summary (1 of 2)**



You have completed this chapter. Let’s summarize the key concepts covered:

* Processes are used to perform various tasks on the system.
* Processes can be single-threaded or multi-threaded.
* Processes can be of different types such as interactive and non-interactive.
* Every process has a unique identifier (PID) to enable the operating system to keep track of it.
* The **nice value**, or **niceness**, can be used to set priority.
* **ps**provides information about the currently running processes.

**Summary (2 of 2)**



* You can use **top** to get constant real-time updates about overall system perfomance as well as information about the processes running on the system.
* Load average indicates the amount of utilization the system is under at particular times.
* Linux supports background and foreground processing for a job.
* **at**executes any non-interactive command at a specified time.
* **cron**is used to schedule tasks that need to be performed at regular intervals.

**Chapter 18: Common Applications**

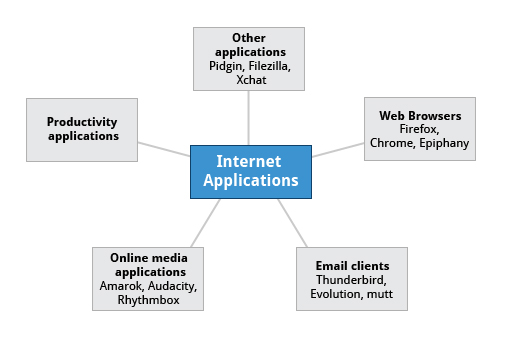
**Learning Objectives**



By the end of this chapter, you should be familiar with common Linux applications, including:

* Internet applications such as browsers, and email programs.
* Office Productivity Suites such as **LibreOffice**.
* Developer tools, such as compilers, debuggers, etc.
* Multimedia applications, such as those for audio and video.
* Graphics editors such as the **GIMP** and other graphics utilities.

**Internet Applications**



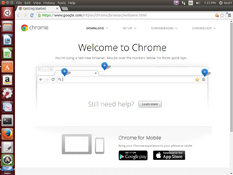
The Internet is a global network that allows users around the world to perform multiple tasks such as searching for data, communicating through emails and online shopping. Obviously, you need to use network-aware applications to take advantage of the Internet. These include:

* Web browsers
* Email clients
* Online media applications
* Other applications

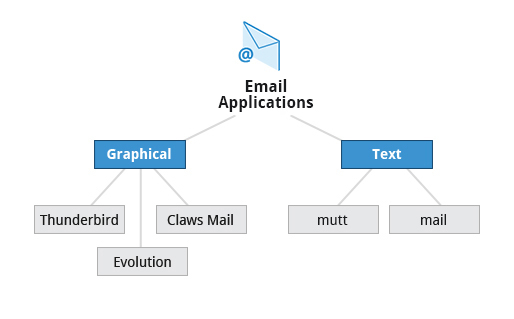
**Web Browsers**

As discussed in the earlier chapter on Network Operations, Linux offers a wide variety of web browsers, both graphical and text based, including:

* **Firefox**
* **Google Chrome**
* **Chromium**
* **Epiphany**
* **Konqueror**
* **w3m**
* **lynx**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch17_screen04_a.jpg) [](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch17_screen04_b.jpg)

**Email Applications**



Email applications allow for sending, receiving, and reading messages over the Internet. Linux systems offer a wide number of **email clients**, both graphical and text-based. In addition many users simply use their browsers to access their email accounts.

Most email clients use the **Internet Message Access Protocol (IMAP)** or the older **Post Office Protocol (POP)** to access emails stored on a remote mail server. Most email applications also display **HTML** (**HyperText Markup Language**) formatted emails that display objects, such as pictures and hyperlinks. The features of advanced email applications include the ability of importing address books/contact lists, configuration information, and emails from other email applications.

Linux supports the following types of email applications:

* Graphical email clients, such as **Thunderbird** (produced by **Mozilla)**, **Evolution**, and **Claws Mail**
* Text mode email clients such as **mutt** and **mail**
* **Other Internet Applications**
* Linux systems provide many other applications for performing Internet-related tasks. These include:
* 

|  |  |
| --- | --- |
| **Application** | **Use** |
| **FileZilla** | Intuitive graphical **FTP** client that supports **FTP**, **Secure File Transfer Protocol (SFTP)**, and **FTP Secured (FTPS).** Used to transfer files to/from **(FTP)** servers. |
| **Pidgin** | To access **GTalk**, **AIM**, **ICQ**, **MSN**, **IRC** and other messaging networks |
| **Ekiga** | To connect to **Voice over Internet Protocol (VoIP)** networks |
| **XChat** | To access **Internet Relay Chat (IRC)** networks |

**Office Applications**



Most day-to-day computer systems have **productivity applications** (sometimes called **office suites**) available or installed ([click here](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/office_suite_software_packages_for_Linux.pdf) for a list of commonly used suites). Each suite is a collection of closely coupled programs used to create and edit different kinds of files such as:

* Text (articles, books, reports etc.)
* Spreadsheet
* Presentation
* Graphical objects

Most Linux distributions offer **LibreOffice**, an open source office suite that started in 2010 and has evolved from **OpenOffice.org.**While other office suites are available as we have listed, **LibreOffice** is the most mature, widely used and intensely developed.

The component applications included in **LibreOffice** are:

|  |  |
| --- | --- |
| **Components of LibreOffice** | **Usage** |
| **Writer** | Word processing |
| **Calc** | Spreadsheets |
| **Impress** | Presentations |
| **Draw** | Create and edit graphics and diagrams |

**Development Applications**

Linux distributions come with a complete set of applications and tools that are needed by those developing or maintaining both user applications and the kernel itself.

These tools are tightly integrated and include:

* Advanced editors customized for programmers' needs, such as **vi** and **emacs**.
* Compilers (such as **gcc** for programs in **C** and **C++**) for every computer language that has ever existed.
* Debuggers such as **gdb** and various graphical front ends to it and many other debugging tools (such as **valgrind**).
* Performance measuring and monitoring programs, some with easy to use graphical interfaces, others more arcane and meant to be used only by serious experienced development engineers.
* Complete Integrated Development Environments (IDE's) such as **Eclipse**, that put all these tools together.

On other operating systems these tools have to be obtained and installed separately, often at a high cost, while on Linux they are all available at no cost through standard package installation systems.

**Sound Players**



Multimedia applications are used to listen to music, view videos, etc, as well as to present and view text and graphics. Linux systems offer a number of **sound player** applications including:

|  |  |
| --- | --- |
| **Application** | **Use** |
| **Amarok** | Mature **MP3** player with a graphical interface, that plays audio and video files, and streams (online audio files). It allows you to create a play list that contains a group of songs, and uses a database to store information about the music collection. |
| **Audacity** | Used to record and edit sounds and can be quickly installed through a package manager. **Audacity** has a simple interface to get you started. |
| **Rhythmbox** | Supports a large variety of digital music sources including streaming Internet audio and podcasts. The application also enables search of particular audio in a library. It supports ‘smart playlists’ with an ‘automatic update’ feature which can revise playlists based on specified selection criteria. |

Of course Linux systems can also connect with commercial online music streaming services such as **Pandora** and **Spotify** through web browsers.

**Movie Players**



**Movie** (video) **players** can portray input from many different sources, either local to the machine or on the Internet.

Linux systems offer a number of movie players including:

* **VLC**
* **MPlayer**
* **Xine**

**Totem**

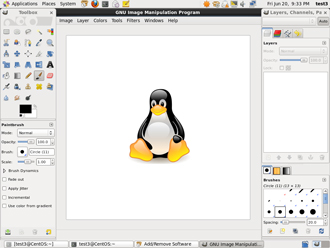
**Movie Editors**



**Movie editors** are used to edit videos or movies. Linux systems offer a number of movie editors including:

|  |  |
| --- | --- |
| **Application** | **Use** |
| **Kino** | Acquire and edit camera streams. **Kino** can merge and separate video clips. |
| **Cinepaint** | Frame-by-frame retouching. **Cinepaint** is used for editing images in a video. |
| **Blender** | Create 3D animation and design. **Blender** is a professional tool that uses modeling as a starting point. There are complex and powerful tools for camera capture, recording, editing, enhancing and creating video, each having its own focus. |
| **Cinelerra** | Capture, compose, and edit audio/video. |
| **FFmpeg** | Record, convert, and stream audio/video. **FFmpeg** is a format converter, among other things, and has other tools such as **ffplay** and **ffserver**. |

**GIMP (GNU Image Manipulation Program)**

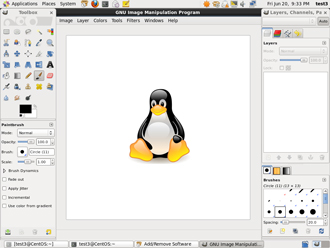
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch17_screen18.jpg)

**Graphic editors** allow you to create, edit, view, and organize images of various formats like Joint Photographic Experts Group (JPEG or JPG), Portable Network Graphics (PNG), Graphics Interchange Format (GIF), and Tagged Image File Format (TIFF).

**GIMP** (**GNU Image Manipulation Program**) is a feature-rich image retouching and editing tool similar to **Adobe Photoshop** and is available on all Linux distributions. Some features of the **GIMP** are:

* It can handle any image file format.
* It has many special purpose plugins and filters.
* It provides extensive information about the image, such as layers, channels, and histograms.

**GIMP (GNU Image Manipulation Program)**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch17_screen18.jpg)

**Graphic editors** allow you to create, edit, view, and organize images of various formats like Joint Photographic Experts Group (JPEG or JPG), Portable Network Graphics (PNG), Graphics Interchange Format (GIF), and Tagged Image File Format (TIFF).

**GIMP** (**GNU Image Manipulation Program**) is a feature-rich image retouching and editing tool similar to **Adobe Photoshop** and is available on all Linux distributions. Some features of the **GIMP** are:

* It can handle any image file format.
* It has many special purpose plugins and filters.
* It provides extensive information about the image, such as layers, channels, and histograms.

**Graphics Utilities**

In addition to the **GIMP**, there are other graphics utilities that help perform various image-related tasks, including:



|  |  |
| --- | --- |
| **Graphic Utility** | **Use** |
| **eog** | **Eye of Gnome (eog)** is an image viewer that provides slide show capability and a few image editing tools, such as rotate and resize. It can also step through the images in a directory with just a click. |
| **Inkscape** | **Inkscape** is an image editor with lots of editing features. It works with layers and transformations of the image. It is sometimes compared to **Adobe FrameMaker.** |
| **convert** | **convert** is a command line tool (part of the **ImageMagick** set of applications) that can modify image files in many ways. The options include file format conversion and numerous image modification options, such as blur, resize, despeckle, etc. |
| **Scribes** | **Scribes** is used for creating documents used for publishing and providing a *What You See Is What You Get (WYSIWYG)* environment. It also provides numerous editing tools. |

**Summary (1 of 2)**



You have completed this chapter. Let’s summarize the key concepts covered:

* Linux offers a wide variety of Internet applications such as web browsers, email clients, online media applications, and others.
* Web browsers supported by Linux can be either graphical or text-based such as **Firefox**, **Google Chrome**, **Epiphany**, **w3m, lynx**and others.
* Linux supports graphical email clients, such as **Thunderbird**, **Evolution**, and **Claws Mail**, and text mode email clients, such as **mutt** and **mail**.
* Linux systems provide many other applications for performing Internet-related tasks, such as **Filezilla**, **XChat**, **Pidgin**, and others.
* Most Linux distributions offer **LibreOffice** to create and edit different kinds of documents.

**Summary (2 of 2)**



* Linux systems offer entire suites of development applications and tools, including compilers and debuggers.
* Linux systems offer a number of **sound players** including **Amarok**, **Audacity**, and **Rhythmbox**.
* Linux systems offer a number of movie players including **VLC**, **MPlayer**, **Xine**, and **Totem**.
* Linux systems offer a number of movie editors including **Kino**, **Cinepaint**, **Blender** among others.
* The **GIMP** (**GNU Image Manipulation Program**) utility is a feature-rich image retouching and editing tool available on all Linux distributions.

Other graphics utilities that help perform various image-related tasks are **eog**, **Inkscape**, **convert**, and **Scribes**.