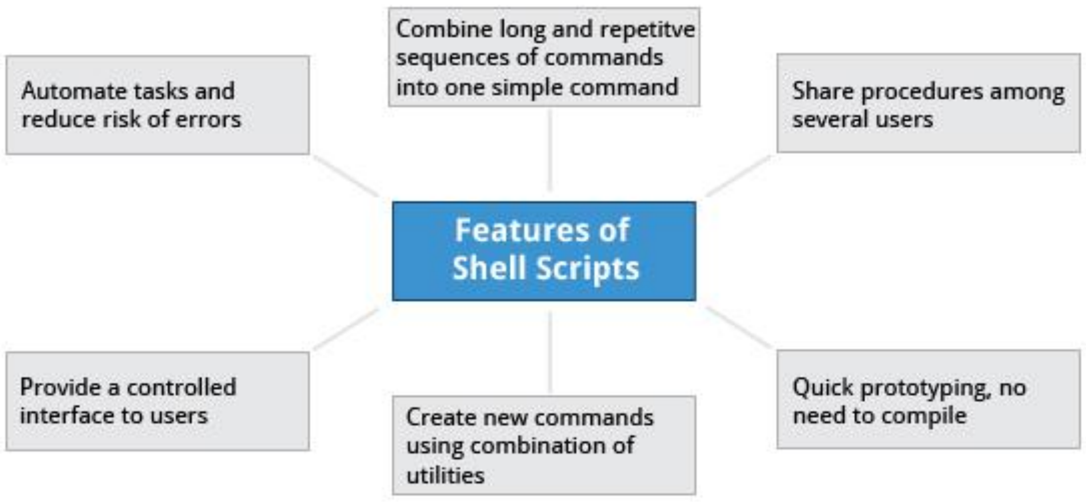
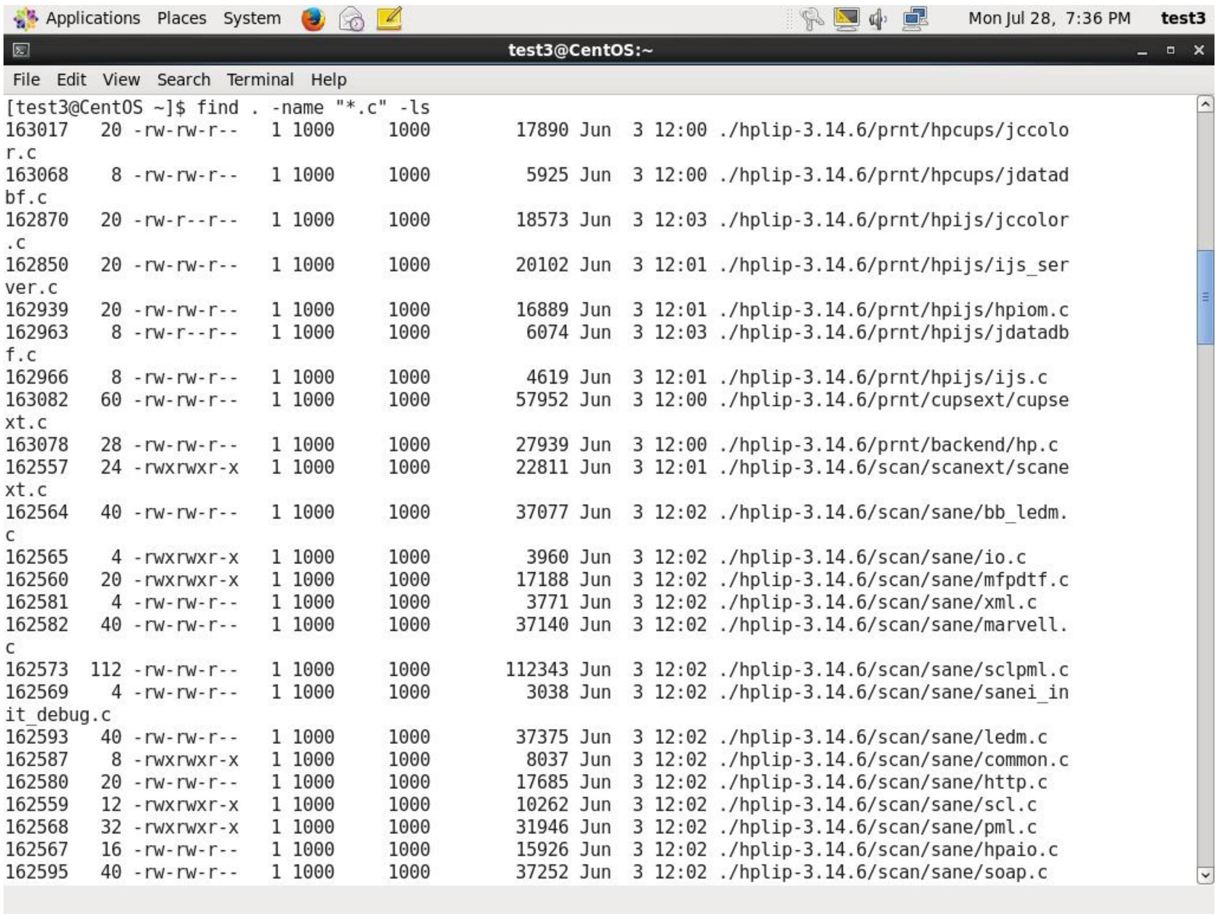
Introduction to Scripts


Automate tasks and reduce risk of errors
Combine long and repetitve sequences of commands into one simple command
Share procedures among several users
Features of Shell Scripts
Provide a controlled interface to users
Create new commands using combination of utilities
Quick prototyping, no need to compile

Suppose you want to look up a filename, check if the associated file exists, and then respond accordingly, displaying a message confirming or not confirming the file's existence. If you only need to do it once, you can just type a sequence of commands at a terminal. However, if you need to do this multiple times, automation is the way to go. In order to automate sets of commands you'll need to learn how to write **shell scripts**, the most common of which are used with **bash**. The graphic illustrates several of the benefits of deploying scripts.

Introduction to Shell Scripts


Applications Places System
Mon Jul 28, 7:36 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

```
[test3@CentOS ~]$ find . -name "*.c" -ls
163017   20 -rw-rw-r--   1 1000     1000        17890 Jun  3 12:00 ./hplip-3.14.6/prnt/hpcups/jccolo
r.c
163068    8 -rw-rw-r--   1 1000     1000         5925 Jun  3 12:00 ./hplip-3.14.6/prnt/hpcups/jdatad
bf.c
162870   20 -rw-r--r--   1 1000     1000        18573 Jun  3 12:03 ./hplip-3.14.6/prnt/hpijs/jccolor
.c
162850   20 -rw-rw-r--   1 1000     1000        20102 Jun  3 12:01 ./hplip-3.14.6/prnt/hpijs/ijs_ser
ver.c
162939   20 -rw-rw-r--   1 1000     1000        16889 Jun  3 12:01 ./hplip-3.14.6/prnt/hpijs/hpiom.c
162963    8 -rw-r--r--   1 1000     1000         6074 Jun  3 12:03 ./hplip-3.14.6/prnt/hpijs/jdatadb
f.c
162966    8 -rw-rw-r--   1 1000     1000         4619 Jun  3 12:01 ./hplip-3.14.6/prnt/hpijs/ijs.c
163082   60 -rw-rw-r--   1 1000     1000        57952 Jun  3 12:00 ./hplip-3.14.6/prnt/cupsext/cupse
xt.c
163078   28 -rw-rw-r--   1 1000     1000        27939 Jun  3 12:00 ./hplip-3.14.6/prnt/backend/hp.c
162557   24 -rwxrwxr-x   1 1000     1000        22811 Jun  3 12:01 ./hplip-3.14.6/scan/scanext/scane
xt.c
162564   40 -rw-rw-r--   1 1000     1000        37077 Jun  3 12:02 ./hplip-3.14.6/scan/sane/bb_ledm.
c
162565    4 -rwxrwxr-x   1 1000     1000         3960 Jun  3 12:02 ./hplip-3.14.6/scan/sane/io.c
162560   20 -rwxrwxr-x   1 1000     1000        17188 Jun  3 12:02 ./hplip-3.14.6/scan/sane/mfpdtf.c
162581    4 -rw-rw-r--   1 1000     1000         3771 Jun  3 12:02 ./hplip-3.14.6/scan/sane/xml.c
162582   40 -rw-rw-r--   1 1000     1000        37140 Jun  3 12:02 ./hplip-3.14.6/scan/sane/marvell.
c
162573  112 -rw-rw-r--   1 1000     1000       112343 Jun  3 12:02 ./hplip-3.14.6/scan/sane/sclpml.c
162569    4 -rw-rw-r--   1 1000     1000         3038 Jun  3 12:02 ./hplip-3.14.6/scan/sane/sanei_in
it_debug.c
162593   40 -rw-rw-r--   1 1000     1000        37375 Jun  3 12:02 ./hplip-3.14.6/scan/sane/ledm.c
162587    8 -rwxrwxr-x   1 1000     1000         8037 Jun  3 12:02 ./hplip-3.14.6/scan/sane/common.c
162580   20 -rw-rw-r--   1 1000     1000        17685 Jun  3 12:02 ./hplip-3.14.6/scan/sane/http.c
162559   12 -rwxrwxr-x   1 1000     1000        10262 Jun  3 12:02 ./hplip-3.14.6/scan/sane/scl.c
162568   32 -rwxrwxr-x   1 1000     1000        31946 Jun  3 12:02 ./hplip-3.14.6/scan/sane/pml.c
162567   16 -rw-rw-r--   1 1000     1000        15926 Jun  3 12:02 ./hplip-3.14.6/scan/sane/hpaio.c
162595   40 -rw-rw-r--   1 1000     1000        37252 Jun  3 12:02 ./hplip-3.14.6/scan/sane/soap.c
```

Remember from our earlier discussion, a **shell** is a command line **interpreter** which provides the user interface for terminal windows. It can also be used to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being directly typed in. For example typing: `find . -name "*.c" -ls` at the command line accomplishes the same thing as executing a script file containing the lines:

```
#!/bin/bash
find . -name "*.c" -ls
```

The `#!/bin/bash` in the first line should be recognized by anyone who has developed any kind of script in UNIX environments. The first line of the script, that starts with `#!`, contains the full path of the command interpreter (in this case `/bin/bash`) that is to be used on the file. As we will see on the next screen, you have a few choices depending upon which scripting language you use.

The command **interpreter** is tasked with executing statements that follow it in the script. Commonly used interpreters include: `/usr/bin/perl`, `/bin/bash`, `/bin/csh`, `/usr/bin/python` and `/bin/sh`.

Typing a long sequence of commands at a terminal window can be complicated, time consuming, and error prone. By deploying shell scripts, using the command-line becomes an efficient and quick way to launch complex sequences of steps. The fact that shell scripts are saved in a file also makes it easy to use them to create new script variations and share standard procedures with several users.

Linux provides a wide choice of shells; exactly what is available on the system is listed in `/etc/shells`. Typical choices are:

```
/bin/sh
/bin/bash
```

```
/bin/tcsh
/bin/csh
/bin/ksh
```

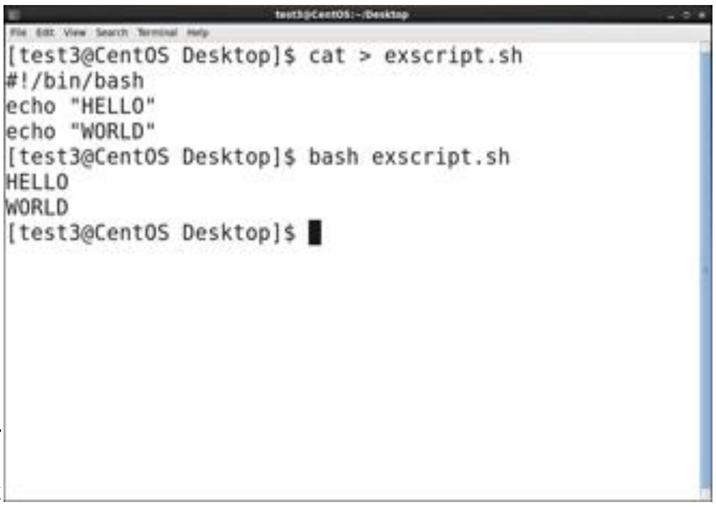
Most Linux users use the default **bash** shell, but those with long UNIX backgrounds with other shells may want to override the default.

bash Scripts

Let's write a simple **bash** script that displays a two-line message on the screen. Either type

```
$ cat > exscript.sh
  #!/bin/bash
  echo "HELLO"
  echo "WORLD"
```

and press **ENTER** and **CTRL-D** to save the file, or just create `exscript.sh` in your favorite text editor. Then, type `chmod +x exscript.sh` to make the file executable. (The `chmod +x` command makes the file executable for all users.) You can then run it by simply typing `./exscript.sh` or by doing:



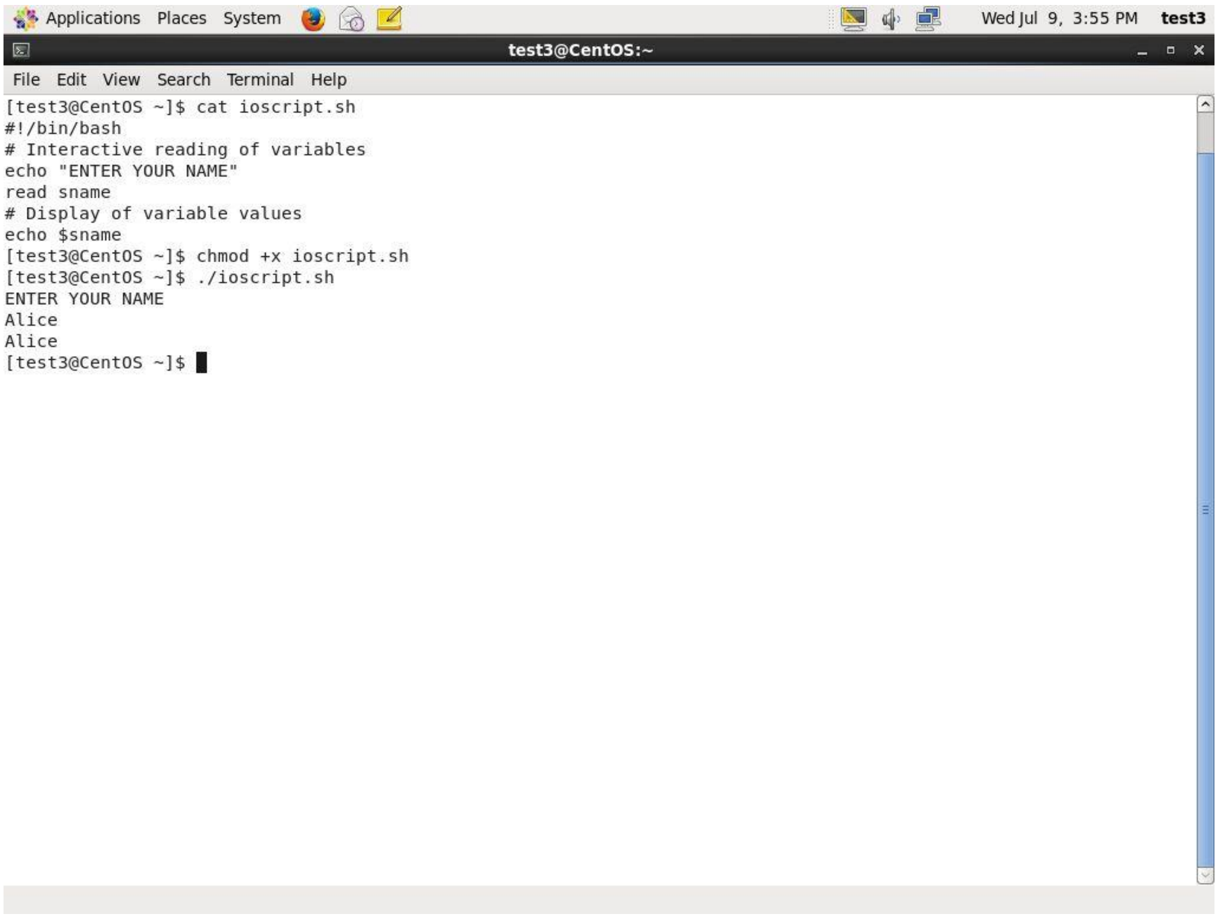
```
[test3@CentOS Desktop]$ cat > exscript.sh
#!/bin/bash
echo "HELLO"
echo "WORLD"
[test3@CentOS Desktop]$ bash exscript.sh
HELLO
WORLD
[test3@CentOS Desktop]$
```

```
$ bash exscript.sh
  HELLO
  WORLD
```

Note if you use the second form, you don't have to make the file executable.

Click the image to view an enlarged version.

Interactive Example Using bash Scripts


Applications Places System
Wed Jul 9, 3:55 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

```
[test3@CentOS ~]$ cat ioscript.sh
#!/bin/bash
# Interactive reading of variables
echo "ENTER YOUR NAME"
read sname
# Display of variable values
echo $sname
[test3@CentOS ~]$ chmod +x ioscript.sh
[test3@CentOS ~]$ ./ioscript.sh
ENTER YOUR NAME
Alice
Alice
[test3@CentOS ~]$
```

Now, let's see how to create a more interactive example using a **bash** script. The user will be prompted to enter a value, which is then displayed on the screen. The value is stored in a temporary variable, `sname`. We can reference the value of a shell variable by using a `$` in front of the variable name, such as `$sname`. To create this script, you need to create a file named `ioscript.sh` in your favorite editor with the following content:

```
    #!/bin/bash
    # Interactive reading of variables
    echo "ENTER YOUR NAME"
    read sname
    # Display of variable values
    echo $sname
```

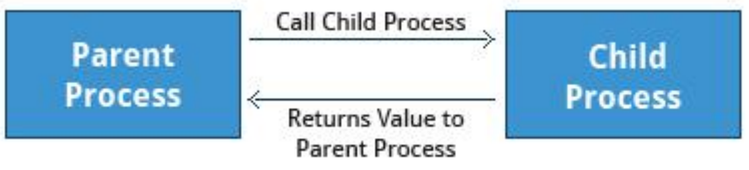
Once again, make it executable by doing `chmod +x ioscript.sh`.

In the above example, when the script `./ioscript.sh` is executed, the user will receive a prompt `ENTER YOUR NAME`. The user then needs to enter a value and press the **Enter** key. The value will then be printed out.

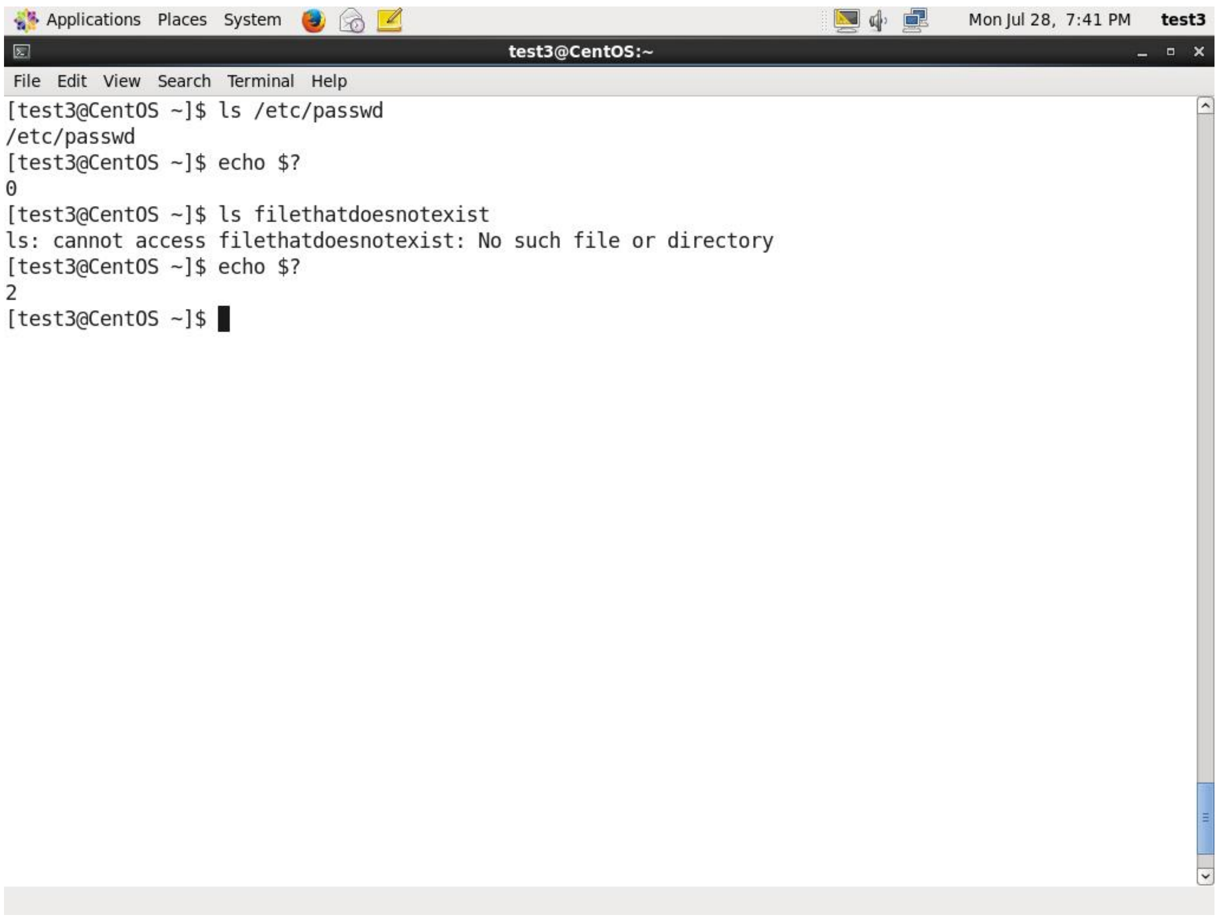
Additional note: The hash-tag/pound-sign/number-sign (`#`) is used to start comments in the script and can be placed anywhere in the line (the rest of the line is considered a comment).

Return Values

All shell scripts generate a **return value** upon finishing execution; the value can be set with the `exit` statement. Return values permit a process to monitor the exit state of another process often in a parent-child relationship. This helps to determine how this process terminated and take any appropriate steps necessary, contingent on success or failure.



Viewing Return Values



```
[test3@CentOS ~]$ ls /etc/passwd
/etc/passwd
[test3@CentOS ~]$ echo $?
0
[test3@CentOS ~]$ ls filethatdoesnotexist
ls: cannot access filethatdoesnotexist: No such file or directory
[test3@CentOS ~]$ echo $?
2
[test3@CentOS ~]$ 
```

As a script executes, one can check for a specific value or condition and return success or failure as the result. By convention, success is returned as 0, and failure is returned as a non-zero value. An easy way to demonstrate success and failure completion is to execute **ls** on a file that exists and one that doesn't, as shown in the following example, where the return value is stored in the environment variable represented by `$?`:

```
$ ls /etc/passwd
/etc/ passwd

$ echo $?
0
```

In this example, the system is able to locate the file `/etc/passwd` and returns a value of `0` to indicate success; the return value is always stored in the `$?` environment variable. Applications often translate these return values into meaningful messages easily understood by the user.

Section2

Basic Syntax and Special Characters

Scripts require you to follow a standard language **syntax**. Rules delineate how to define variables and how to construct and format allowed statements, etc. The table lists some special character usages within **bash** scripts:

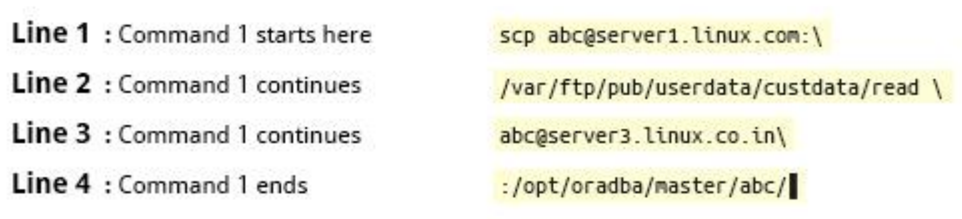
Character	Description
`#`	Used to add a comment, **except** when used as `\#`, or as `#!` when starting a script
`\`	Used at the end of a line to indicate continuation on to the next line
`;`	Used to interpret what follows as a new command
`$`	Indicates what follows is a variable

Note that when `#` is inserted at the beginning of a line of commentary, the whole line is ignored.

```
# This line will not get executed.
```

Splitting Long Commands Over Multiple Lines

Users sometimes need to combine several commands and statements and even conditionally execute them based on the behaviour of operators used in between them. This method is called **chaining of commands**.



```
Line 1 : Command 1 starts here      scp abc@server1.linux.com:\
Line 2 : Command 1 continues        /var/ftp/pub/userdata/custdata/read \
Line 3 : Command 1 continues        abc@server3.linux.co.in\
Line 4 : Command 1 ends             :/opt/oradba/master/abc/
```

The **concatenation operator** (\) is used to concatenate large commands over several lines in the shell.

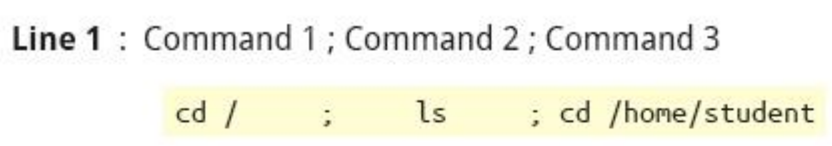
For example, you want to copy the file **/var/ftp/pub/userdata/custdata/read** from **server1.linux.com** to the **/opt/oradba/master/abc** directory on **server3.linux.co.in**. To perform this action, you can write the command using the \ operator as:

```
scp abc@server1.linux.com:\
/var/ftp/pub/userdata/custdata/read \
abc@server3.linux.co.in:\
/opt/oradba/master/abc/
```

The command is divided into multiple lines to make it look readable and easier to understand. The \ operator at the end of each line combines the commands from multiple lines and executes it as one single command.

Putting Multiple Commands on a Single Line

Sometimes you may want to group multiple commands on a single line. The `;` (semicolon) character is used to separate these commands and execute them sequentially as if they had been typed on separate lines.



```
Line 1 : Command 1 ; Command 2 ; Command 3
         cd /      ;   ls      ; cd /home/student
```

The three commands in the following example will all execute even if the ones preceding them fail:

```
$ make ; make install ; make clean
```

However, you may want to abort subsequent commands if one fails. You can do this using the `&&` (and) operator as in:

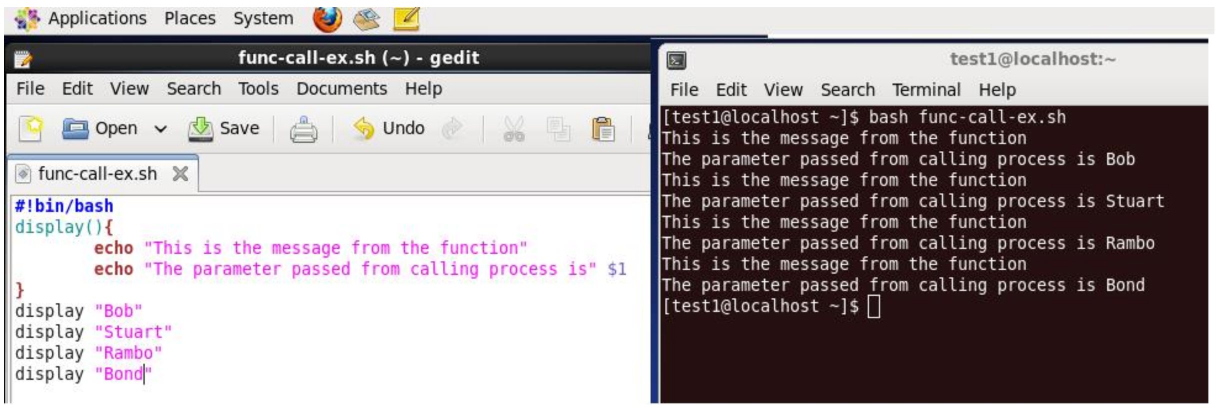
```
$ make && make install && make clean
```

If the first command fails the second one will never be executed. A final refinement is to use the `||` (or) operator as in:

```
$ cat file1 || cat file2 || cat file3
```

In this case, you proceed until something succeeds and then you stop executing any further steps.

Functions


Applications Places System
func-call-ex.sh (~) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
func-call-ex.sh
test1@localhost:~
File Edit View Search Terminal Help

```
#!bin/bash
display(){
        echo "This is the message from the function"
        echo "The parameter passed from calling process is" $1
}
display "Bob"
display "Stuart"
display "Rambo"
display "Bond"
```

```
[test1@localhost ~]$ bash func-call-ex.sh
This is the message from the function
The parameter passed from calling process is Bob
This is the message from the function
The parameter passed from calling process is Stuart
This is the message from the function
The parameter passed from calling process is Rambo
This is the message from the function
The parameter passed from calling process is Bond
[test1@localhost ~]$
```

A **function** is a code block that implements a set of operations. Functions are useful for executing procedures multiple times perhaps with varying input variables. Functions are also often called **subroutines.** Using functions in scripts requires two steps:

1. Declaring a function
2. Calling a function

The function declaration requires a name which is used to invoke it. The proper syntax is:

```
function_name () {
   command...
}
```

For example, the following function is named `display`:

```
display () {
   echo "This is a sample function"
}
```

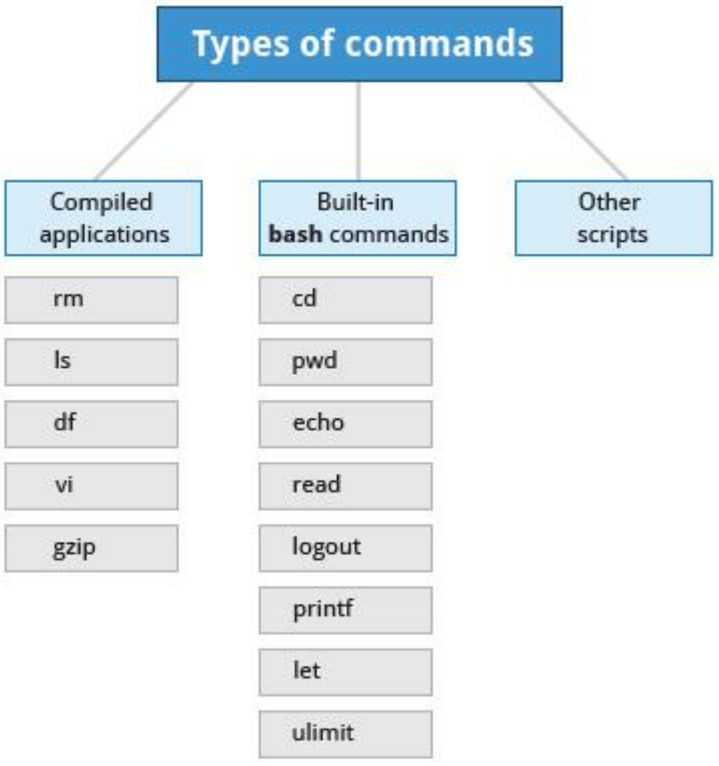
The function can be as long as desired and have many statements. Once defined, the function can be called later as many times as necessary. In the full example shown in the figure, we are also showing an often-used refinement: how to pass an argument to the function. The first argument can be referred to as `$1`, the second as `$2`, etc.

Click the image to view an enlarged version.

Built-in Shell Commands

Shell scripts are used to execute sequences of commands and other types of statements. Commands can be divided into the following categories:

- Compiled applications
- Built-in **bash** commands
- Other scripts

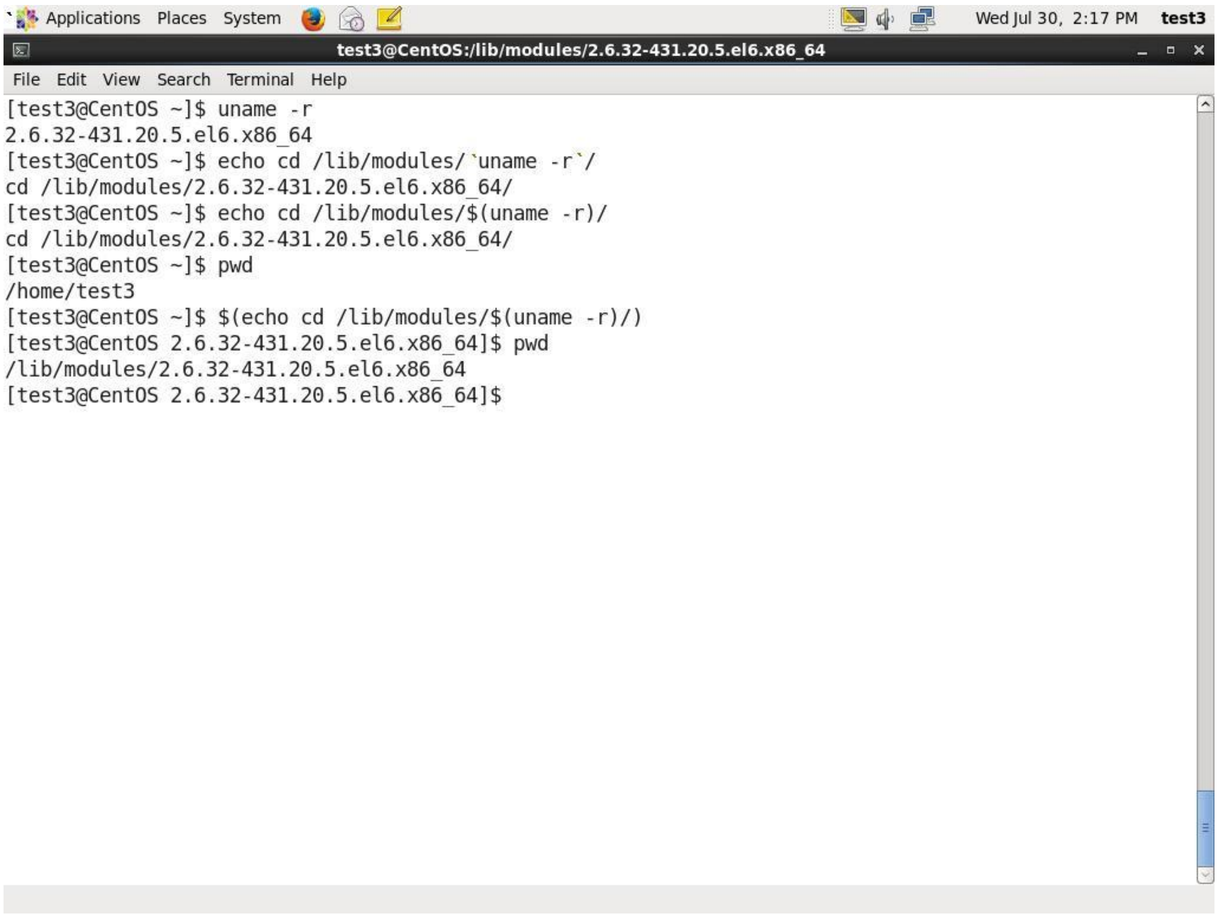


Compiled applications are binary executable files that you can find on the filesystem. The shell script always has access to compiled applications such as **rm**, **ls**, **df**, **vi**, and **gzip**.

bash has many **built-in** commands which can only be used to display the output within a terminal shell or shell script. Sometimes these commands have the same name as executable programs on the system, such as **echo** which can lead to subtle problems. **bash** built-in commands include and `cd`, `pwd`, `echo`, `read`, `logout`, `printf`, `let`, and `ulimit`.

A complete list of **bash** built-in commands can be found in the **bash man** page, or by simply typing `help`.

Command Substitution


Applications Places System
Wed Jul 30, 2:17 PM test3
test3@CentOS:/lib/modules/2.6.32-431.20.5.el6.x86_64
File Edit View Search Terminal Help

```
[test3@CentOS ~]$ uname -r
2.6.32-431.20.5.el6.x86_64
[test3@CentOS ~]$ echo cd /lib/modules/`uname -r`/
cd /lib/modules/2.6.32-431.20.5.el6.x86_64/
[test3@CentOS ~]$ echo cd /lib/modules/$(uname -r)/
cd /lib/modules/2.6.32-431.20.5.el6.x86_64/
[test3@CentOS ~]$ pwd
/home/test3
[test3@CentOS ~]$ $(echo cd /lib/modules/$(uname -r)/)
[test3@CentOS 2.6.32-431.20.5.el6.x86_64]$ pwd
/lib/modules/2.6.32-431.20.5.el6.x86_64
[test3@CentOS 2.6.32-431.20.5.el6.x86_64]$
```

At times, you may need to substitute the result of a command as a portion of another command. It can be done in two ways:

- By enclosing the inner command with backticks (`)
- By enclosing the inner command in `$( )`

No matter the method, the innermost command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution was done.

Virtually any command can be executed this way. Both of these methods enable command substitution; however, the `$( )` method allows command nesting. New scripts should always use this more modern method. For example:

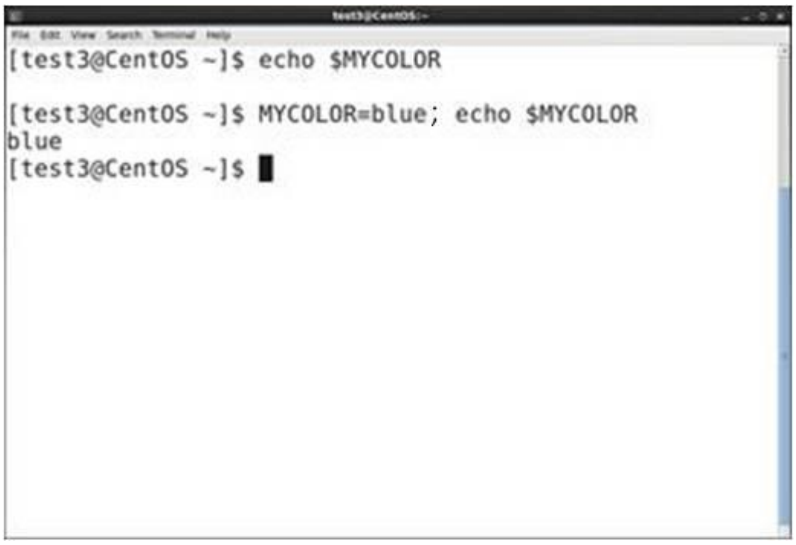
```
$ cd /lib/modules/$(uname -r)/
```

In the above example, the output of the command `"uname -r"` becomes the argument for the `cd` command.

Click the image to view an enlarged version.

Environment Variables

Almost all scripts use **variables** containing a value, which can be used anywhere in the script. These variables can either be user or system defined. Many applications use such **environment variables** (covered in the "User Environment" chapter) for supplying inputs, validation, and controlling behaviour.



```
[test3@CentOS ~]$ echo $MYCOLOR

[test3@CentOS ~]$ MYCOLOR=blue; echo $MYCOLOR
blue
[test3@CentOS ~]$ 
```

Some examples of standard environment variables are `HOME`, `PATH`, and `HOST`. When referenced, environment variables must be prefixed with the `$` symbol as in `$HOME`. You can view and set the value of environment variables. For example, the following command displays the value stored in the `PATH` variable:

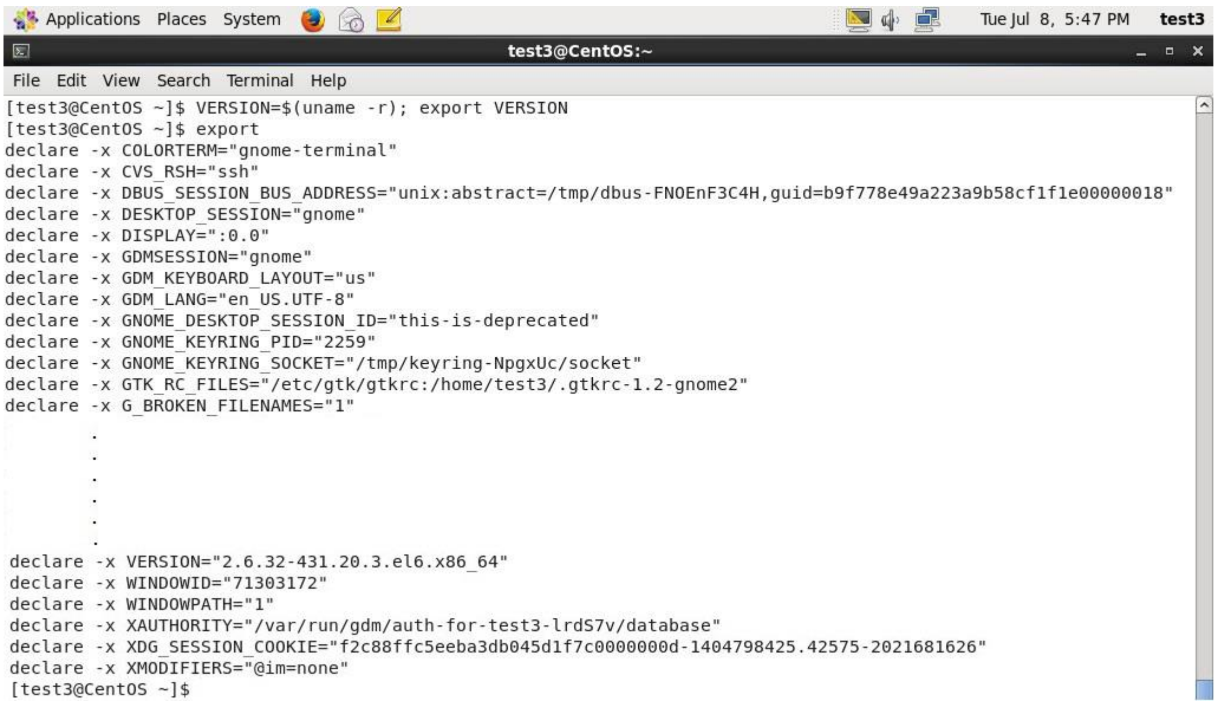
```
$ echo $PATH
```

However, no prefix is required when setting or modifying the variable value. For example, the following command sets the value of the `MYCOLOR` variable to blue:

```
$ MYCOLOR=blue
```

You can get a list of environment variables with the **env**, **set**, or **printenv** commands.

Exporting Variables



```
[test3@CentOS ~]$ VERSION=$(uname -r); export VERSION
[test3@CentOS ~]$ export
declare -x COLORTERM="gnome-terminal"
declare -x CVS_RSH="ssh"
declare -x DBUS_SESSION_BUS_ADDRESS="unix:abstract=/tmp/dbus-FNOEnF3C4H,guid=b9f778e49a223a9b58cf1f1e00000018"
declare -x DESKTOP_SESSION="gnome"
declare -x DISPLAY=":0.0"
declare -x GDMSESSION="gnome"
declare -x GDM_KEYBOARD_LAYOUT="us"
declare -x GDM_LANG="en_US.UTF-8"
declare -x GNOME_DESKTOP_SESSION_ID="this-is-deprecated"
declare -x GNOME_KEYRING_PID="2259"
declare -x GNOME_KEYRING_SOCKET="/tmp/keyring-NpgxUc/socket"
declare -x GTK_RC_FILES="/etc/gtk/gtkrc:/home/test3/.gtkrc-1.2-gnome2"
declare -x G_BROKEN_FILENAMES="1"
        .
        .
        .
        .
        .
        .
declare -x VERSION="2.6.32-431.20.3.el6.x86_64"
declare -x WINDOWID="71303172"
declare -x WINDOWPATH="1"
declare -x XAUTHORITY="/var/run/gdm/auth-for-test3-lrdS7v/database"
declare -x XDG_SESSION_COOKIE="f2c88ffc5eeba3db045d1f7c0000000d-1404798425.42575-2021681626"
declare -x XMODIFIERS="@im=none"
[test3@CentOS ~]$
```

By default, the variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be promoted to environment variables using the **export** statement as in:

```
export VAR=value
```

or

```
VAR=value ; export VAR
```

While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, but only copied.

Script Parameters

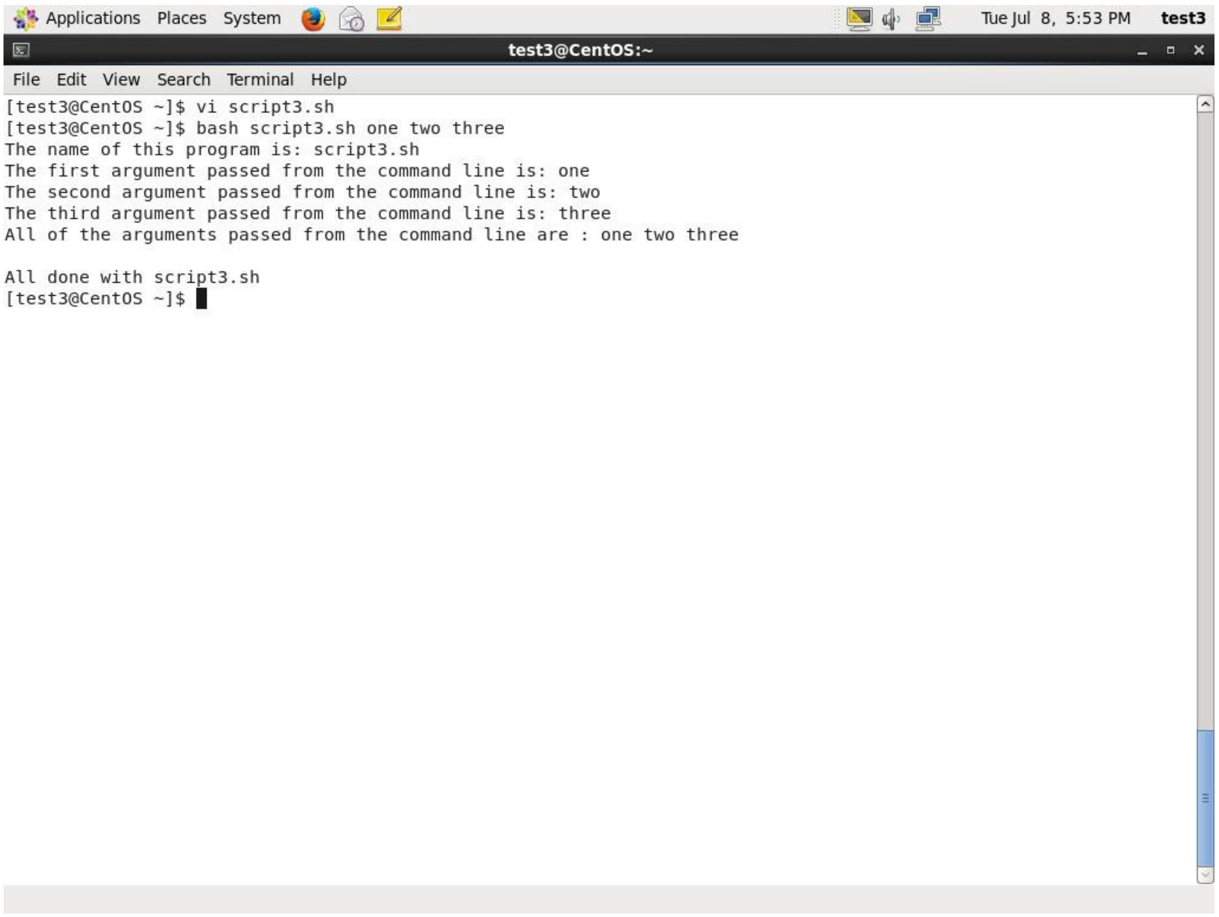
Users often need to pass parameter values to a script, such as a filename, date, etc. Scripts will take different paths or arrive at different values according to the parameters (command arguments) that are passed to them. These values can be text or numbers as in:

```
$ ./script.sh /tmp
$ ./script.sh 100 200
```

Within a script, the parameter or an argument is represented with a `$` and a number. The table lists some of these parameters.

Parameter	Meaning
$0	Script name
$1	First parameter
$2, $3, etc.	Second, third parameter, etc.
$*	All parameters
$#	Number of arguments

Using Script Parameters


Applications Places System
Tue Jul 8, 5:53 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

```
[test3@CentOS ~]$ vi script3.sh
[test3@CentOS ~]$ bash script3.sh one two three
The name of this program is: script3.sh
The first argument passed from the command line is: one
The second argument passed from the command line is: two
The third argument passed from the command line is: three
All of the arguments passed from the command line are : one two three

All done with script3.sh
[test3@CentOS ~]$ 
```

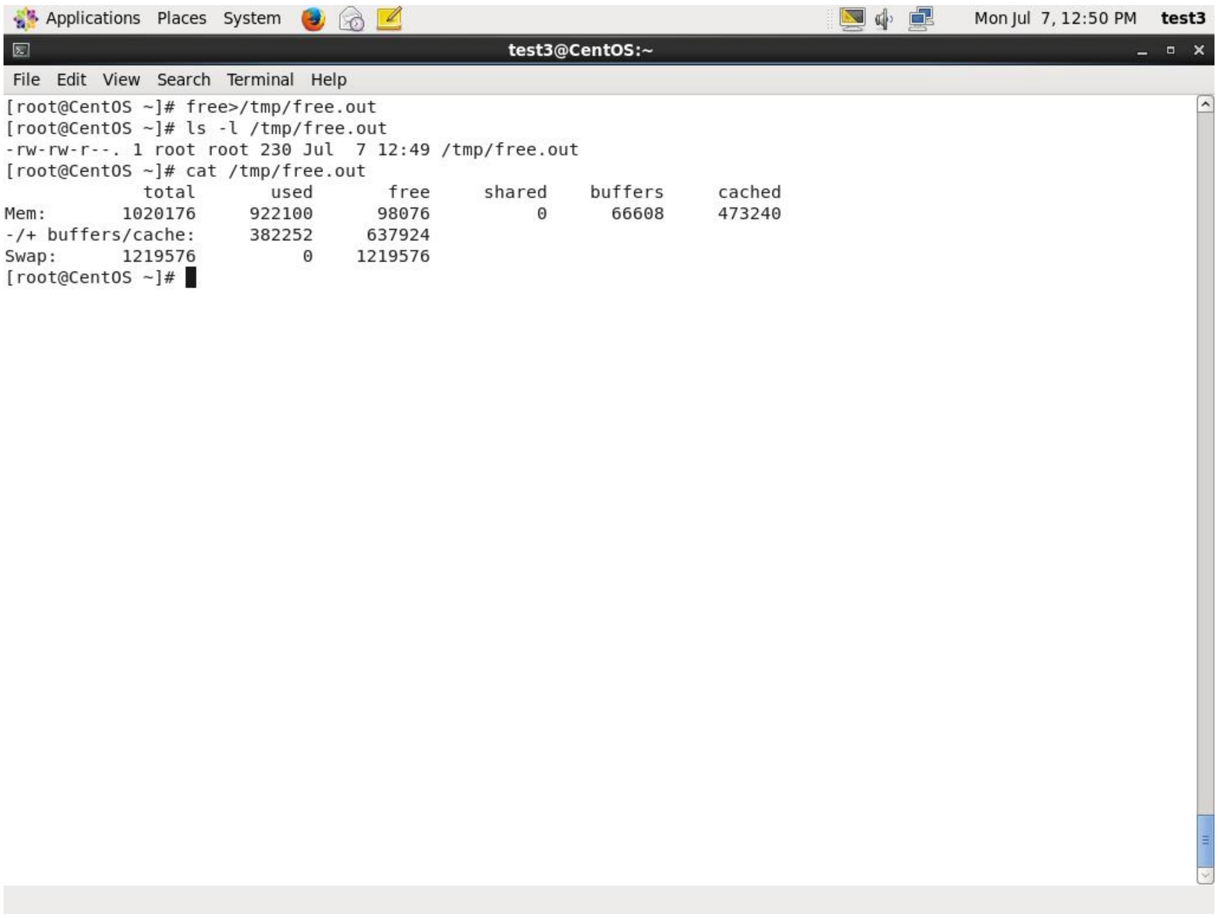
Using your favorite text editor, create a new script file named **script3.sh** with the following contents:

```
#!/bin/bash
echo The name of this program is: $0
echo The first argument passed from the command line is: $1
echo The second argument passed from the command line is: $2
echo The third argument passed from the command line is: $3
echo All of the arguments passed from the command line are : $*
echo
echo All done with $0
```

Make the script executable with **chmod +x**. Run the script giving it three arguments as in: `script3.sh one two three`, and the script is processed as follows:

`$0` prints the script name: `script3.sh`
`$1` prints the first parameter: `one`
`$2` prints the second parameter: `two`
`$3` prints the third parameter: `three`
`$*` prints all parameters: `one two three`
The final statement becomes: `All done with script3.sh`

Output Redirection


Applications Places System
Mon Jul 7, 12:50 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

```
[root@CentOS ~]# free>/tmp/free.out
[root@CentOS ~]# ls -l /tmp/free.out
-rw-rw-r--. 1 root root 230 Jul  7 12:49 /tmp/free.out
[root@CentOS ~]# cat /tmp/free.out
             total       used       free     shared    buffers     cached
Mem:       1020176     922100      98076          0      66608     473240
-/+ buffers/cache:     382252     637924
Swap:      1219576          0    1219576
[root@CentOS ~]# 
```

Most operating systems accept input from the keyboard and display the output on the terminal. However, in shell scripting you can send the output to a file. The process of diverting the output to a file is called output **redirection**.

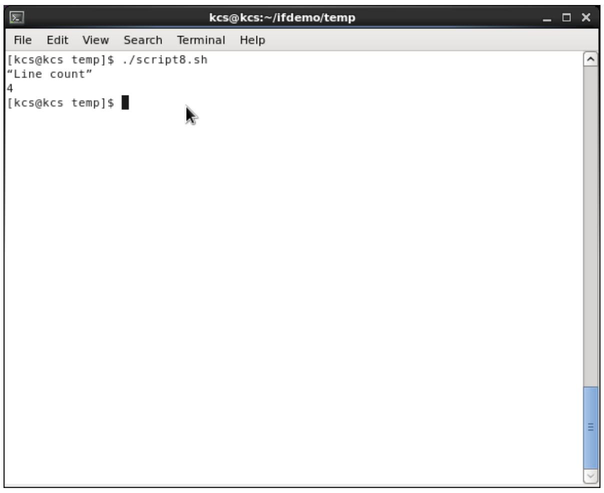
The > character is used to write output to a file. For example, the following command sends the output of **free** to the file `/tmp/free.out`:

```
$ free > /tmp/free.out
```

To check the contents of the `/tmp/free.out` file, at the command prompt type `cat /tmp/free.out`.

Two > characters (>>) will append output to a file if it exists, and act just like > if the file does not already exist.

Input Redirection



```
[kcs@kcs temp]$ ./script8.sh
“Line count”
4
[kcs@kcs temp]$ 
```

Just as the output can be redirected **to** a file, the input of a command can be read **from** a file. The process of reading input from a file is called input redirection and uses the `<` character. If you create a file called `script8.sh` with the following contents:

```
#!/bin/bash
echo “Line count”
wc -l < /temp/free.out
```

and then execute it with `chmod +x script8.sh ; ./script8.sh`, it will count the number of lines from the **`/temp/free.out`** file and display the results.

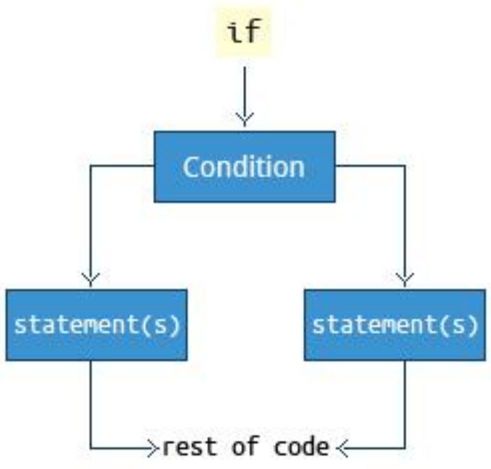
Section3

The if Statement

Conditional decision making using an `if` statement, is a basic construct that any useful programming or scripting language must have.

When an `if` statement is used, the ensuing actions depend on the evaluation of specified conditions such as:

- Numerical or string comparisons
- Return value of a command (0 for success)
- File existence or permissions



In compact form, the syntax of an `if` statement is:

```
if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi
```

A more general definition is:

```
if condition
then
       statements
```

```
else
       statements
fi
```

```
#!/bin/bash
file=$1
if [ -f $file ]
then
 echo -e "plik $file istnieje"
else
 echo -e "plik $file" nie istnieje"
fi
```

```
$ touch plik1 plik2 plik3 plik5
$ ls plik*
plik1  plik2  plik3  plik5
$ bash if-demo.sh plik1
plik plik1 istnieje
$ bash if-demo.sh plik4
plik plik4 nie istnieje
```

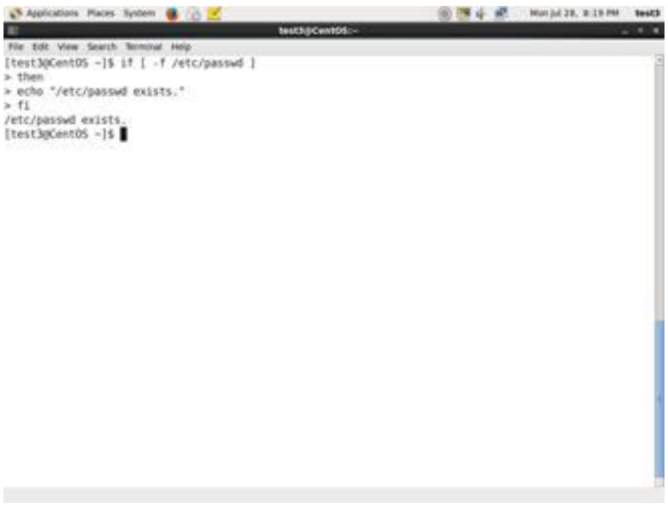
Nested if

```
#!/bin/bash
echo "Wpisz pierwsza liczbe"
read arg1
echo „Wpisz druga liczbe"
read arg2
echo „1. Dodawanie"
echo „2. Odejnowanie"
echo „3. Mnozenie"
echo „Wprowadz numer operacji: 1, 2 lub 3"
read op
if [ $op -eq 1 ]
  echo "Wynik dodawania: " $(($l1+$l2))
else

  if [ $op -eq 2 ]
  then
    echo "wynik odejmowania: " $(($l1-$l2))
  else
    if [ $op -eq 3 ]
    then
      echo "Wynik mnozenia: " $(($l1*$l2))
    else
      echo "Nie prawidlowa operacja"

    fi
  fi
fi
```

Konstrukcja elif

```
#!/bin/bash
echo „podaj liczbe”
read liczba
if [ $liczba -eq 100]
then
  echo “liczba wynosi 100”
elif [ $liczba -gt 100 ]
  echo „liczba jest wieksza od 100”
else
  echo „liczba jest mniejsza od 100”
fi
```

Using the if Statement


Applications Places System
test3@CentOS:~
File Edit View Search Terminal Help

```
[test3@CentOS ~]$ if [ -f /etc/passwd ]
> then
> echo "/etc/passwd exists."
> fi
/etc/passwd exists.
[test3@CentOS ~]$
```

The following `if` statement checks for the `/etc/passwd` file, and if the file is found it displays the message `/etc/passwd exists.`:

```
if [ -f /etc/passwd ]
then
    echo "/etc/passwd exists."
fi
```

Notice the use of the square brackets (`[]`) to delineate the test condition. There are many other kinds of tests you can perform, such as checking whether two numbers are equal to, greater than, or less than each other and make a decision accordingly; we will discuss these other tests.

In modern scripts you may see doubled brackets as in `[[ -f /etc/passwd ]]`. This is not an error. It is never wrong to do so and it avoids some subtle problems such as referring to an empty environment variable without surrounding it in double quotes; we won't talk about this here.

You can use the `if` statement to test for file attributes such as:

- File or directory existence
- Read or write permission
- Executable permission

For example, in the following example:

```
if [ -f /etc/passwd ] ; then
    ACTION
fi
```

the `if` statement checks if the file `/etc/passwd` is a regular file.
Note the very common practice of putting **“; then”** on the same line as the **if** statement.

bash provides a set of **file conditionals**, that can used with the `if` statement, including:

Condition	Meaning
`-e file`	Check if the file exists.
`-d file`	Check if the file is a directory.
`-f file`	Check if the file is a regular file (i.e., not a symbolic link, device node, directory, etc.)
`-s file`	Check if the file is of non-zero size.
`-g file`	Check if the file has `sgid` set.
`-u file`	Check if the file has `suid` set.
`-r file`	Check if the file is readable.
`-w file`	Check if the file is writable.
`-x file`	Check if the file is executable.

You can view the full list of file conditions using the command `man 1 test`.

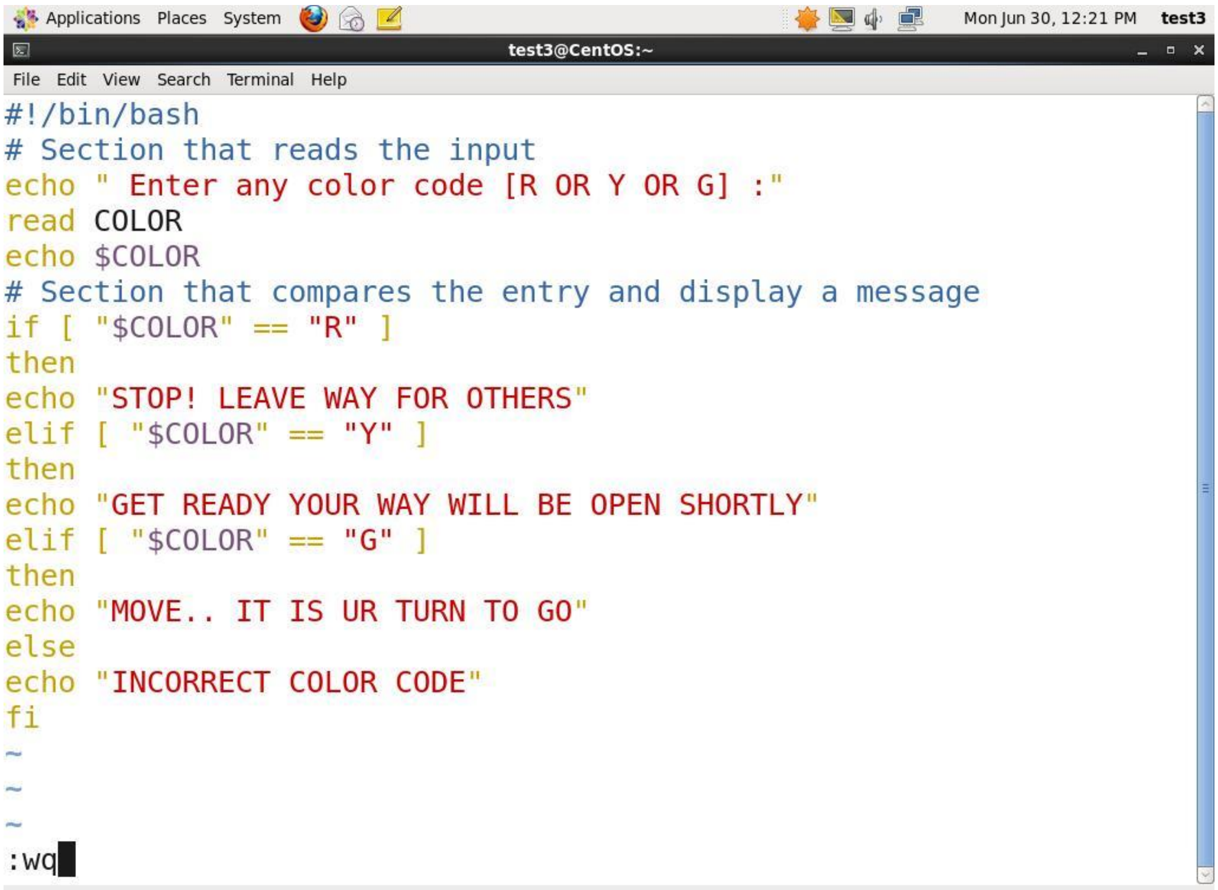
Example of Testing of Strings

You can use the `if` statement to compare strings using the operator `==` (two equal signs). The syntax is as follows:

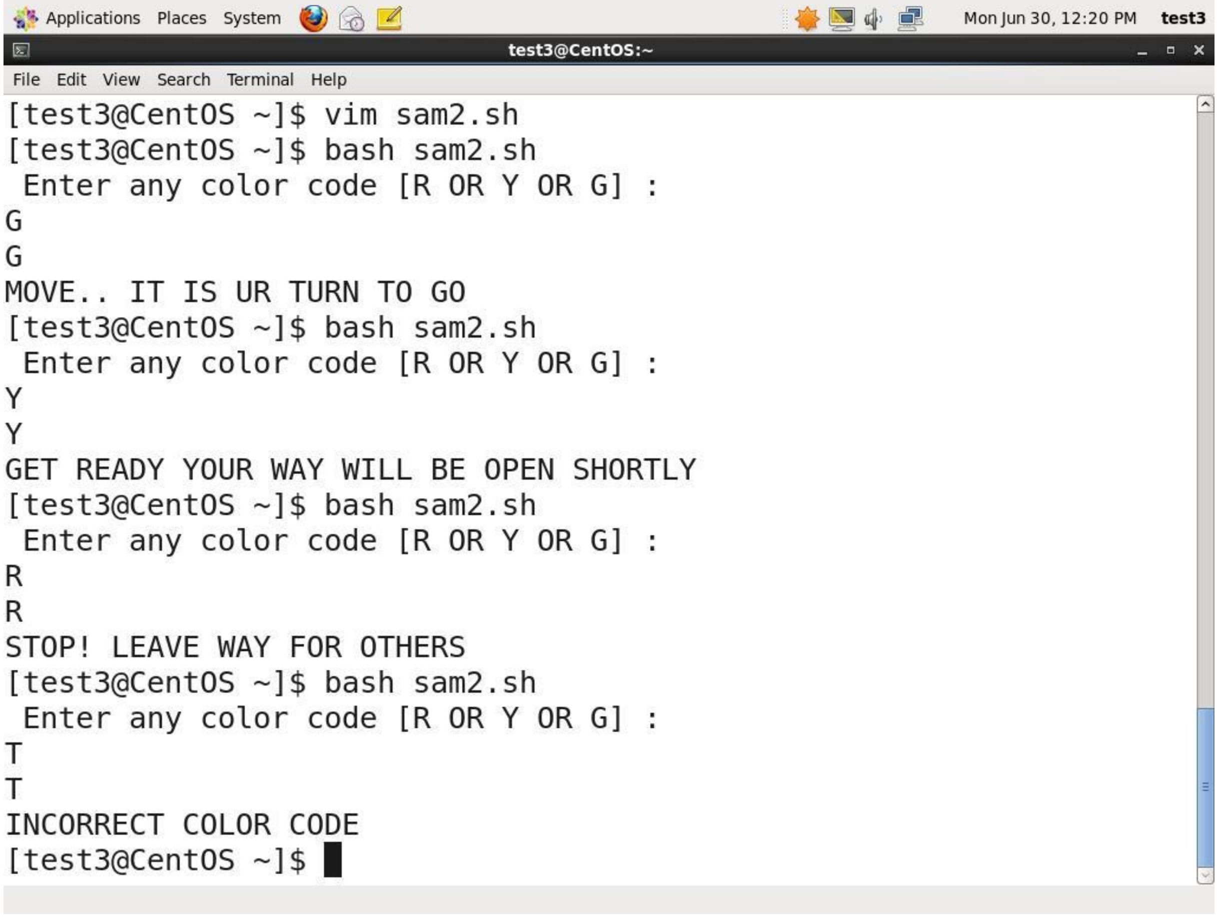
```
if [ string1 == string2 ] ; then
   ACTION
fi
```

Let's now consider an example of testing strings.

In the example illustrated here, the `if` statement is used to compare the input provided by the user and accordingly display the result.


Applications Places System
Mon Jun 30, 12:21 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

```
#!/bin/bash
# Section that reads the input
echo " Enter any color code [R OR Y OR G] :"
read COLOR
echo $COLOR
# Section that compares the entry and display a message
if [ "$COLOR" == "R" ]
then
echo "STOP! LEAVE WAY FOR OTHERS"
elif [ "$COLOR" == "Y" ]
then
echo "GET READY YOUR WAY WILL BE OPEN SHORTLY"
elif [ "$COLOR" == "G" ]
then
echo "MOVE.. IT IS UR TURN TO GO"
else
echo "INCORRECT COLOR CODE"
fi
~
~
~
:wq
```


Applications Places System
Mon Jun 30, 12:20 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

```
[test3@CentOS ~]$ vim sam2.sh
[test3@CentOS ~]$ bash sam2.sh
 Enter any color code [R OR Y OR G] :
G
G
MOVE.. IT IS UR TURN TO GO
[test3@CentOS ~]$ bash sam2.sh
 Enter any color code [R OR Y OR G] :
Y
Y
GET READY YOUR WAY WILL BE OPEN SHORTLY
[test3@CentOS ~]$ bash sam2.sh
 Enter any color code [R OR Y OR G] :
R
R
STOP! LEAVE WAY FOR OTHERS
[test3@CentOS ~]$ bash sam2.sh
 Enter any color code [R OR Y OR G] :
T
T
INCORRECT COLOR CODE
[test3@CentOS ~]$
```

Click the image to view an enlarged version

Numerical Tests

You can use specially defined operators with the `if` statement to compare numbers. The various operators that are available are listed in the table.

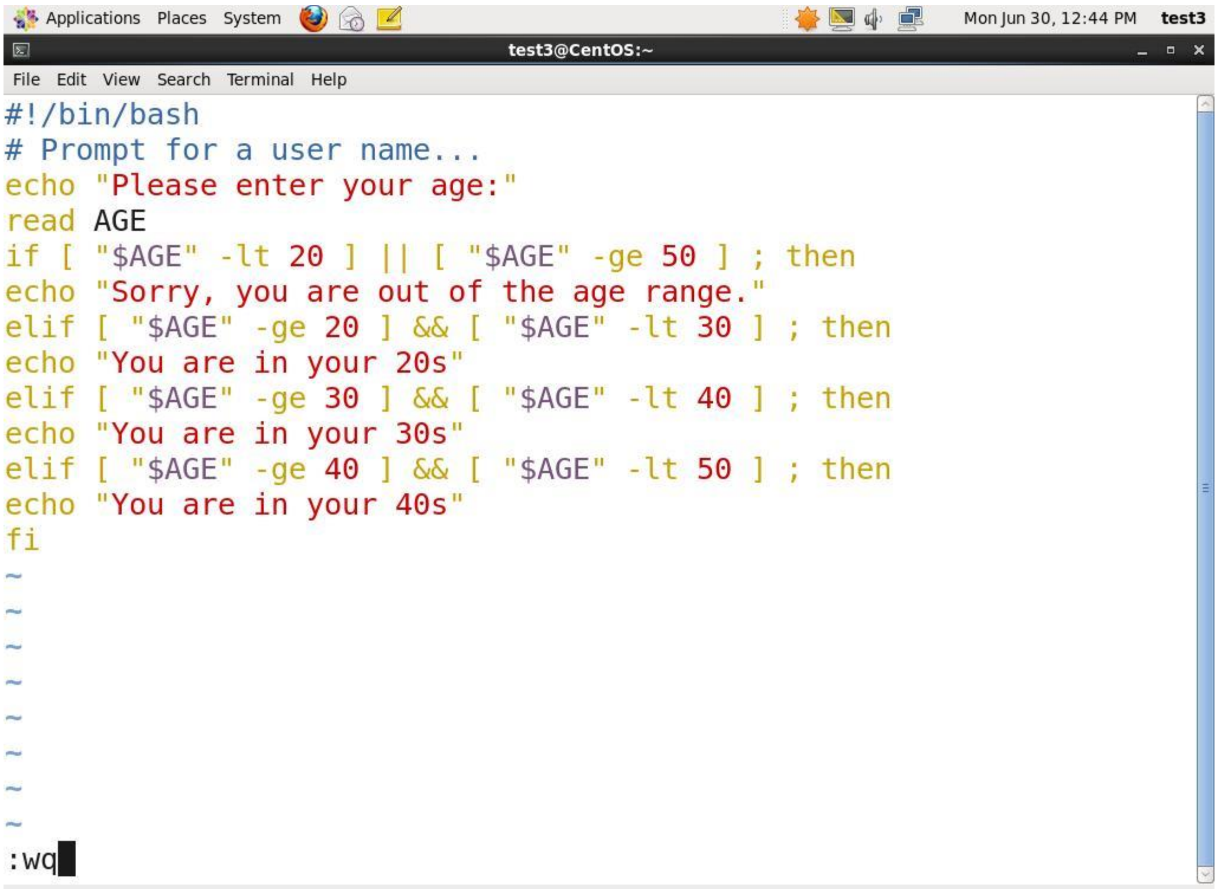
Operator	Meaning
`-eq`	Equal to
`-ne`	Not equal to
`-gt`	Greater than
`-lt`	Less than
`-ge`	Greater than or equal to
`-le`	Less than or equal to

The syntax for comparing numbers is as follows:

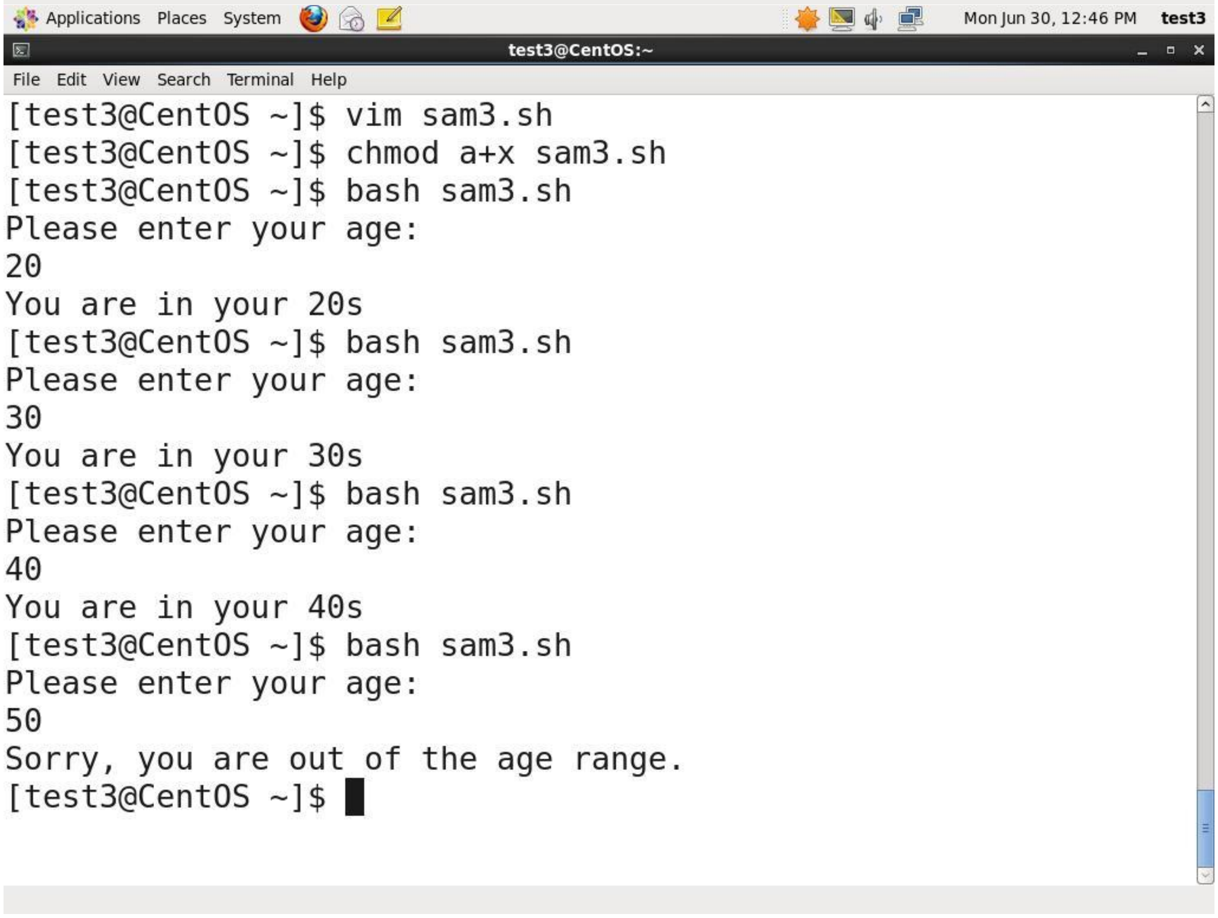
```
exp1 -op exp2
```

Example of Testing for Numbers

Let us now consider an example of comparing numbers using the various operators:


Applications Places System
Mon Jun 30, 12:44 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

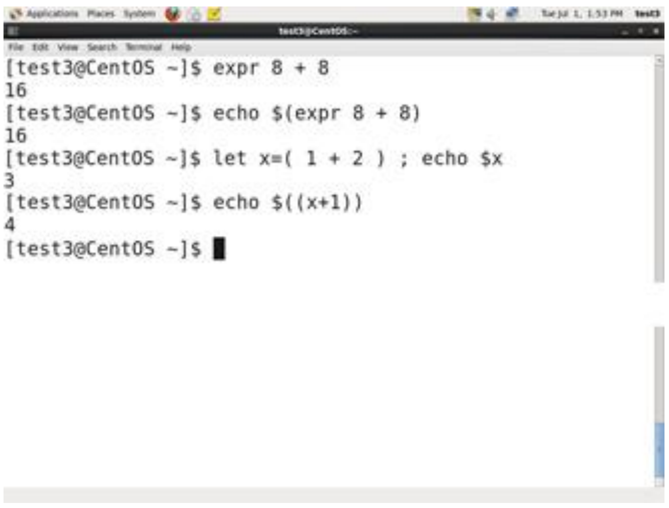
```
#!/bin/bash
# Prompt for a user name...
echo "Please enter your age:"
read AGE
if [ "$AGE" -lt 20 ] || [ "$AGE" -ge 50 ] ; then
echo "Sorry, you are out of the age range."
elif [ "$AGE" -ge 20 ] && [ "$AGE" -lt 30 ] ; then
echo "You are in your 20s"
elif [ "$AGE" -ge 30 ] && [ "$AGE" -lt 40 ] ; then
echo "You are in your 30s"
elif [ "$AGE" -ge 40 ] && [ "$AGE" -lt 50 ] ; then
echo "You are in your 40s"
fi
~
~
~
~
~
~
~
~
:wq
```


Applications Places System
Mon Jun 30, 12:46 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

```
[test3@CentOS ~]$ vim sam3.sh
[test3@CentOS ~]$ chmod a+x sam3.sh
[test3@CentOS ~]$ bash sam3.sh
Please enter your age:
20
You are in your 20s
[test3@CentOS ~]$ bash sam3.sh
Please enter your age:
30
You are in your 30s
[test3@CentOS ~]$ bash sam3.sh
Please enter your age:
40
You are in your 40s
[test3@CentOS ~]$ bash sam3.sh
Please enter your age:
50
Sorry, you are out of the age range.
[test3@CentOS ~]$
```

Click the image to view an enlarged version.

Arithmetic Expressions



```
[test3@CentOS ~]$ expr 8 + 8
16
[test3@CentOS ~]$ echo $(expr 8 + 8)
16
[test3@CentOS ~]$ let x=( 1 + 2 ) ; echo $x
3
[test3@CentOS ~]$ echo $((x+1))
4
[test3@CentOS ~]$
```

Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

- Using the **expr** utility: **expr** is a standard but somewhat deprecated program. The syntax is as follows:

  ```
  expr 8 + 8
  echo $(expr 8 + 8)
  ```
- Using the `$((...))` syntax: This is the built-in shell format. The syntax is as follows:

  ```
  echo $((x+1))
  ```
- Using the built-in shell command `let`. The syntax is as follows:

  ```
  let x=( 1 + 2 ); echo $x
  ```

In modern shell scripts the use of **expr** is better replaced with `var=$((...))`

Click the image to view an enlarged version.

Section 1

String Manipulation

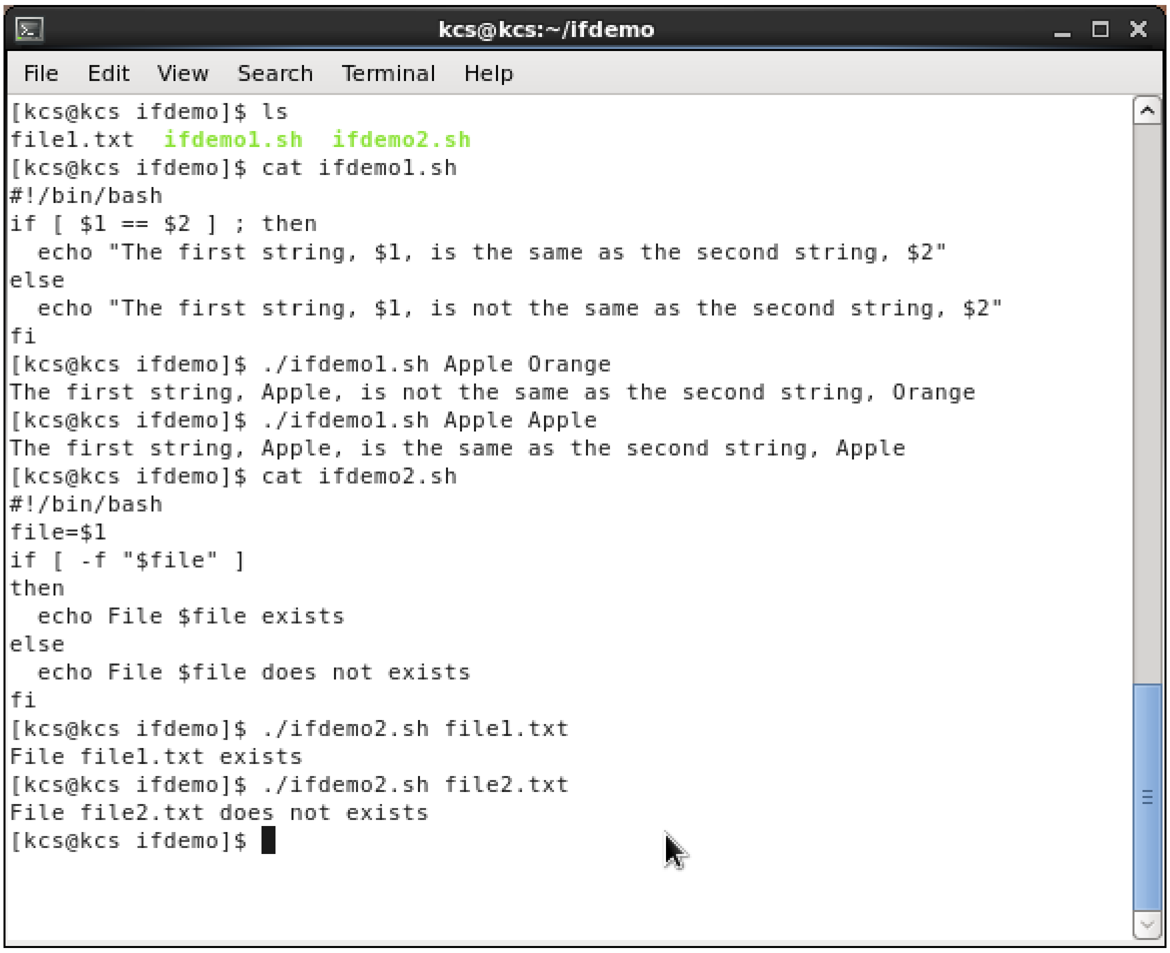
Let's go deeper and find out how to work with strings in scripts.

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



```
[kcs@kcs ifdemo]$ ls
file1.txt  ifdemo1.sh  ifdemo2.sh
[kcs@kcs ifdemo]$ cat ifdemo1.sh
#!/bin/bash
if [ $1 == $2 ] ; then
  echo "The first string, $1, is the same as the second string, $2"
else
  echo "The first string, $1, is not the same as the second string, $2"
fi
[kcs@kcs ifdemo]$ ./ifdemo1.sh Apple Orange
The first string, Apple, is not the same as the second string, Orange
[kcs@kcs ifdemo]$ ./ifdemo1.sh Apple Apple
The first string, Apple, is the same as the second string, Apple
[kcs@kcs ifdemo]$ cat ifdemo2.sh
#!/bin/bash
file=$1
if [ -f "$file" ]
then
  echo File $file exists
else
  echo File $file does not exists
fi
[kcs@kcs ifdemo]$ ./ifdemo2.sh file1.txt
File file1.txt exists
[kcs@kcs ifdemo]$ ./ifdemo2.sh file2.txt
File file2.txt does not exists
[kcs@kcs ifdemo]$ 
```

See the screen shot above.

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

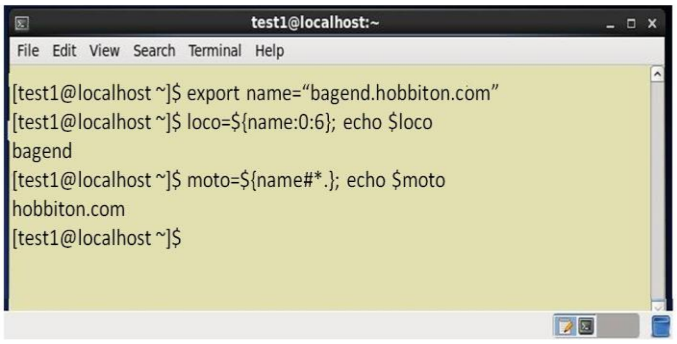
In the second example, we pass in a file name and see if that file exists in the current directory or not.

Click the image to view an enlarged version.

Parts of a String

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.



```
[test1@localhost ~]$ export name="bagend.hobbiton.com"
[test1@localhost ~]$ loco=${name:0:6}; echo $loco
bagend
[test1@localhost ~]$ moto=${name#*.}; echo $moto
hobbiton.com
[test1@localhost ~]$
```

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

Sekcja3

Boolean Expressions

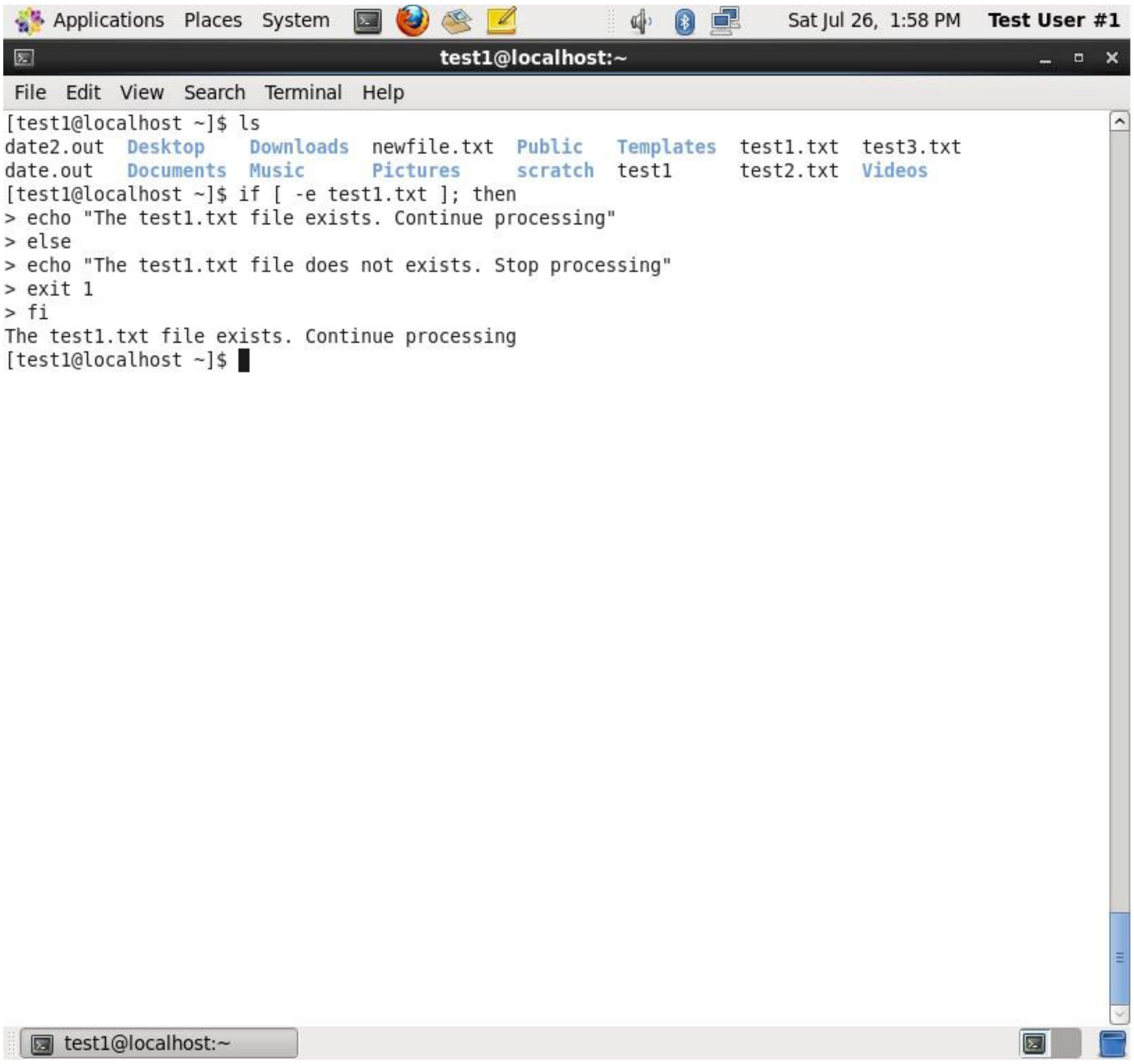
Boolean expressions evaluate to either **TRUE** or **FALSE**, and results are obtained using the various Boolean operators listed in the table.

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


Applications Places System
Sat Jul 26, 1:58 PM
Test User #1
test1@localhost:~
File Edit View Search Terminal Help

```
[test1@localhost ~]$ ls
date2.out  Desktop    Downloads  newfile.txt  Public   Templates  test1.txt  test3.txt
date.out   Documents  Music      Pictures     scratch  test1      test2.txt  Videos
[test1@localhost ~]$ if [ -e test1.txt ]; then
> echo "The test1.txt file exists. Continue processing"
> else
> echo "The test1.txt file does not exists. Stop processing"
> exit 1
> fi
The test1.txt file exists. Continue processing
[test1@localhost ~]$
```

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> ]
```

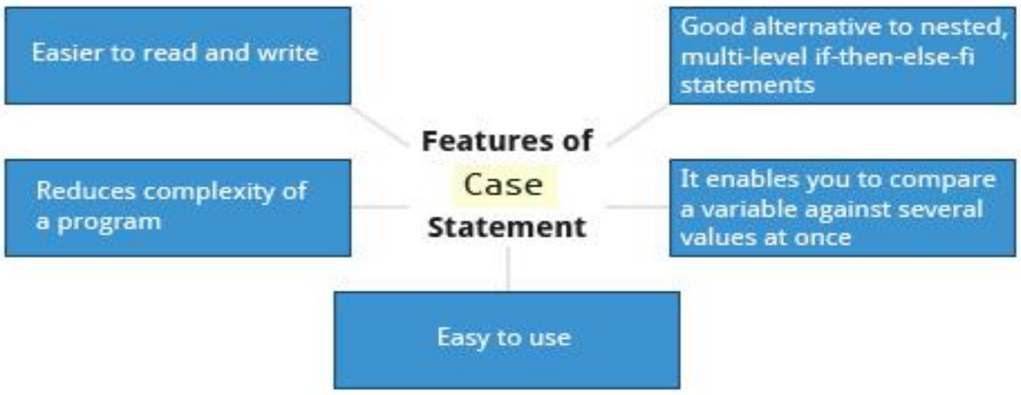
Similarly, to check if the value of `number1` is greater than the value of `number2`, use the following conditional test:

```
[ $number1 -gt $number2 ]
```

The operator `-gt` returns **TRUE** if `number1` is greater than `number2`.

sekcja3

The case Statement


Easier to read and write
Good alternative to nested, multi-level if-then-else-fi statements
Features of Case Statement
Reduces complexity of a program
It enables you to compare a variable against several values at once
Easy to use

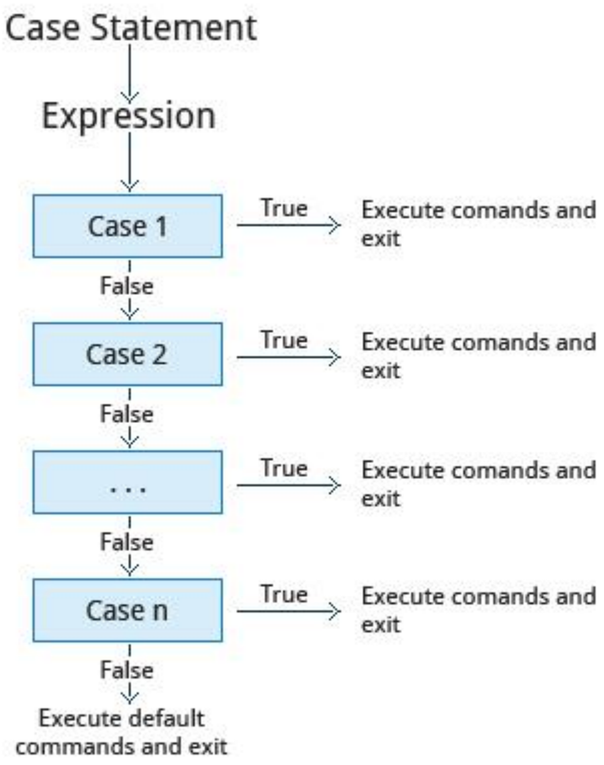
The `case` statement is used in scenarios where the actual value of a variable can lead to different execution paths. `case` statements are often used to handle command-line options.

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

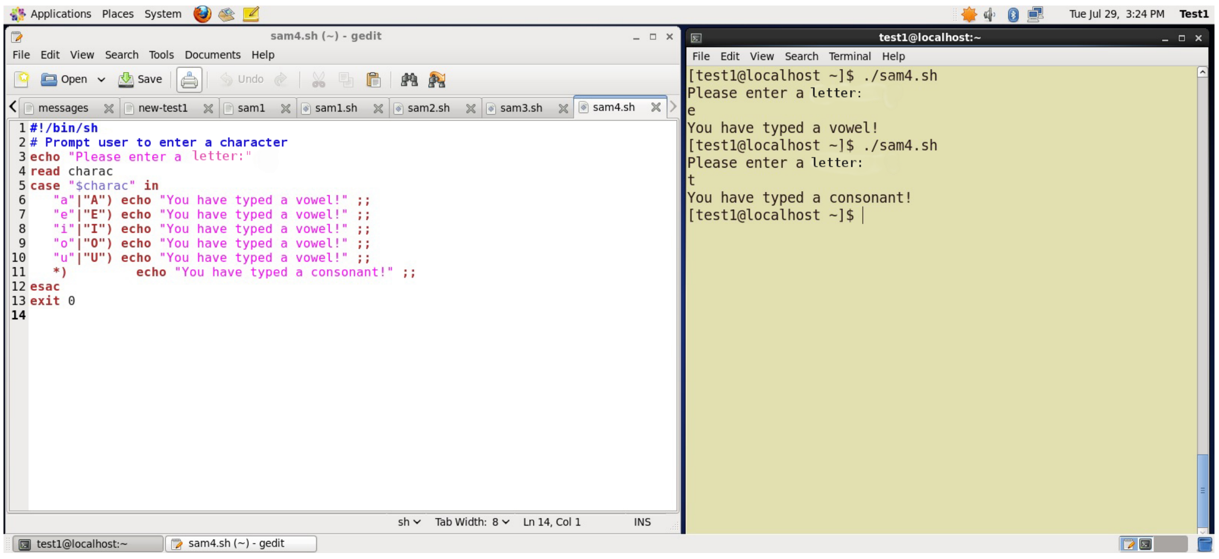
- It is easier to read and write.
- It is a good alternative to nested, multi-level `if-then-else-fi` code blocks.
- It enables you to compare a variable against several values at once.
- It reduces the complexity of a program.

- **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
```



Example of the case Statement



```
#!/bin/sh
# Prompt user to enter a character
echo "Please enter a letter:"
read charac
case "$charac" in
   "a"|"A") echo "You have typed a vowel!" ;;
   "e"|"E") echo "You have typed a vowel!" ;;
   "i"|"I") echo "You have typed a vowel!" ;;
   "o"|"O") echo "You have typed a vowel!" ;;
   "u"|"U") echo "You have typed a vowel!" ;;
   *)          echo "You have typed a consonant!" ;;
esac
exit 0
```

```
[test1@localhost ~]$ ./sam4.sh
Please enter a letter:
e
You have typed a vowel!
[test1@localhost ~]$ ./sam4.sh
Please enter a letter:
t
You have typed a consonant!
[test1@localhost ~]$
```

Here's an example of a `case` statement, please click on the image to open it in a new tab.

Sekcja4

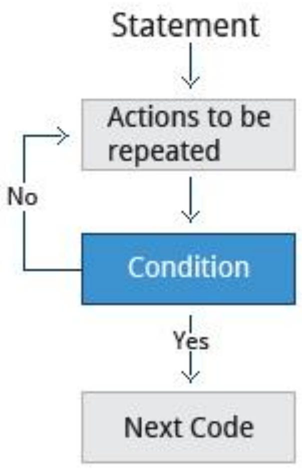
Looping Constructs

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.

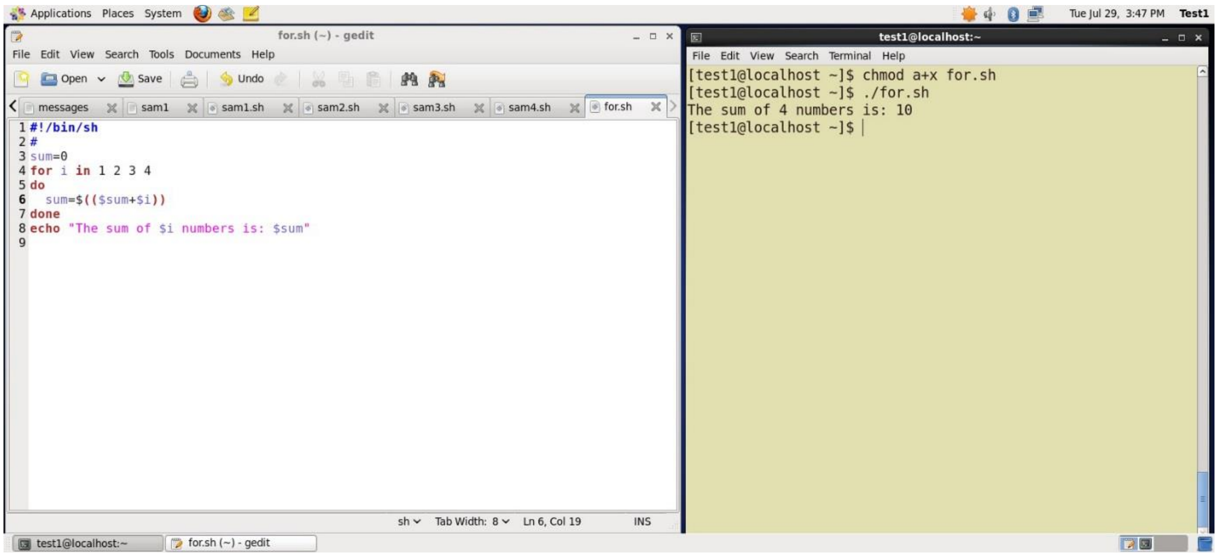
Three type of loops are often used in most programming languages:

- `for`
- `while`
- `until`

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



The 'for' Loop



```
#!/bin/sh
#
sum=0
for i in 1 2 3 4
do
  sum=$(($sum+$i))
done
echo "The sum of $i numbers is: $sum"
```

```
[test1@localhost ~]$ chmod a+x for.sh
[test1@localhost ~]$ ./for.sh
The sum of 4 numbers is: 10
[test1@localhost ~]$
```

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

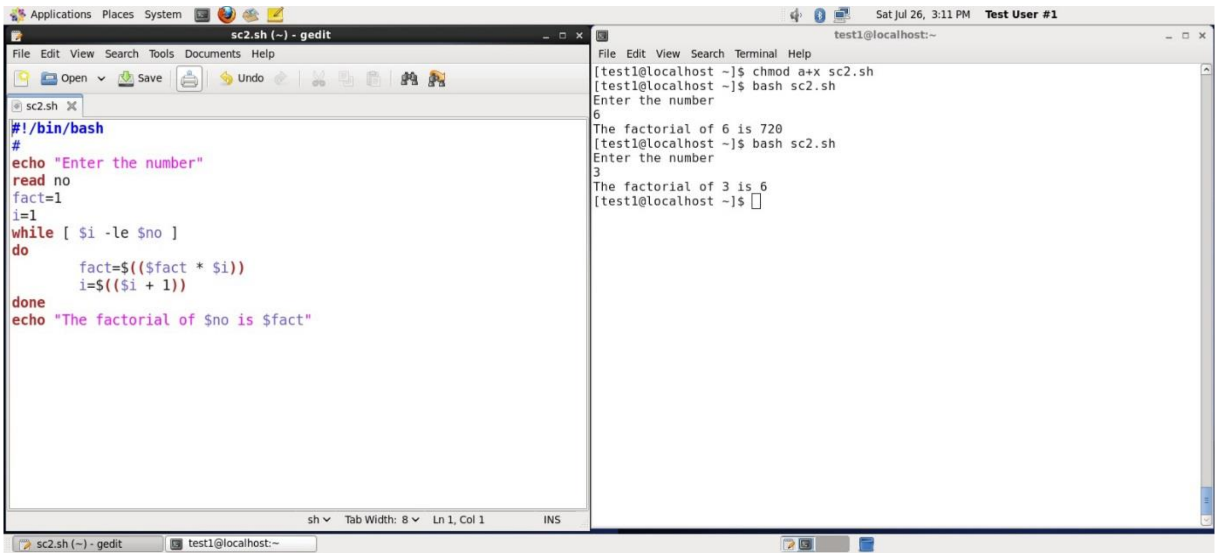
```
for variable-name in list
do
     execute one iteration for each item in the
      list until the list is finished
done
```

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.

Click the image to view an enlarged version.

The while Loop



```
#!/bin/bash
#
echo "Enter the number"
read no
fact=1
i=1
while [ $i -le $no ]
do
        fact=$(($fact * $i))
        i=$(($i + 1))
done
echo "The factorial of $no is $fact"
```

```
[test1@localhost ~]$ chmod a+x sc2.sh
[test1@localhost ~]$ bash sc2.sh
Enter the number
6
The factorial of 6 is 720
[test1@localhost ~]$ bash sc2.sh
Enter the number
3
The factorial of 3 is 6
[test1@localhost ~]$
```

The `while` loop repeats a set of statements as long as the control command returns true. The syntax is:

```
while condition is true
do
```

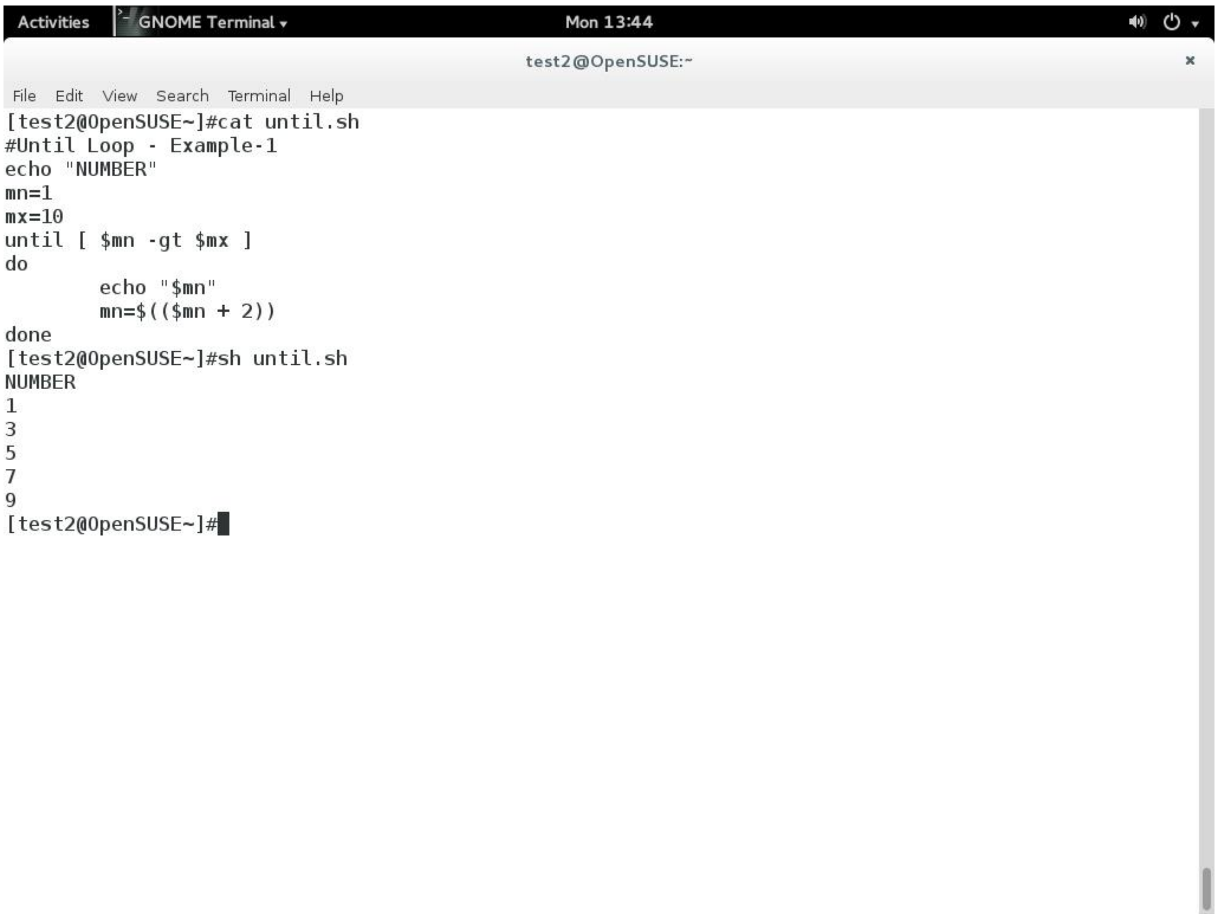
```
    Commands for execution
    ----
done
```

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.

Click the image to view an enlarged version.

The until loop



```
[test2@OpenSUSE~]#cat until.sh
#Until Loop - Example-1
echo "NUMBER"
mn=1
mx=10
until [ $mn -gt $mx ]
do
        echo "$mn"
        mn=$(($mn + 2))
done
[test2@OpenSUSE~]#sh until.sh
NUMBER
1
3
5
7
9
[test2@OpenSUSE~]#
```

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.

Section5

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

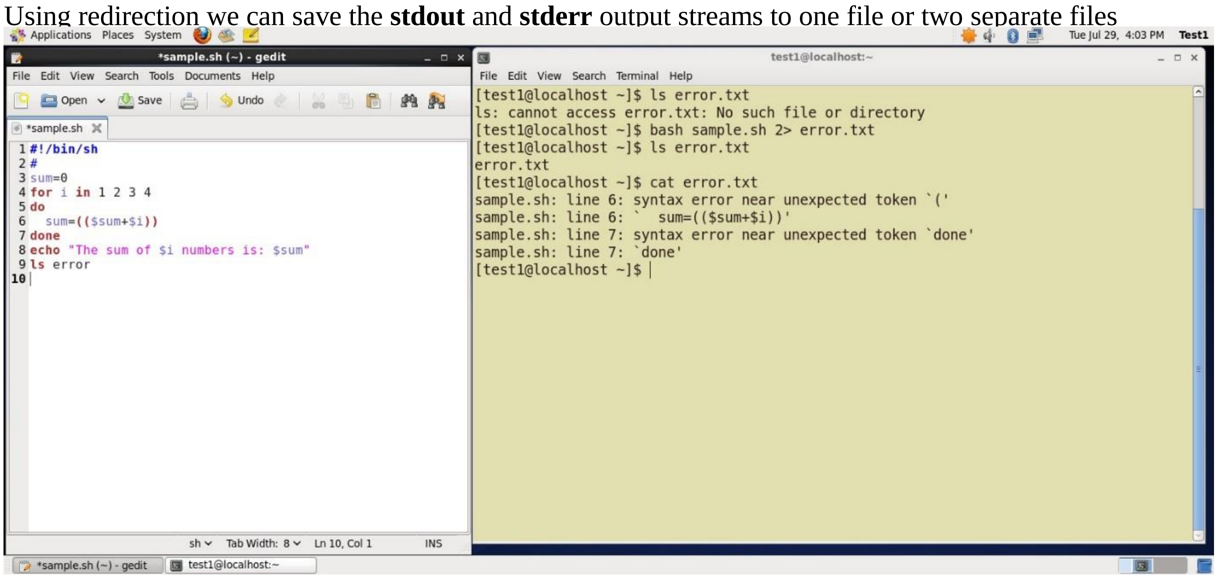
Before fixing an error (or bug), it is vital to know its source.

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
```
- **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



```
1 #!/bin/sh
2 #
3 sum=0
4 for i in 1 2 3 4
5 do
6   sum=(($sum+$i))
7 done
8 echo "The sum of $i numbers is: $sum"
9 ls error
10
```

```
[test1@localhost ~]$ ls error.txt
ls: cannot access error.txt: No such file or directory
[test1@localhost ~]$ bash sample.sh 2> error.txt
[test1@localhost ~]$ ls error.txt
error.txt
[test1@localhost ~]$ cat error.txt
sample.sh: line 6: syntax error near unexpected token `('
sample.sh: line 6: `  sum=(($sum+$i))'
sample.sh: line 7: syntax error near unexpected token `done'
sample.sh: line 7: `done'
[test1@localhost ~]$
```

- 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).

Czesc3

Section1

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.

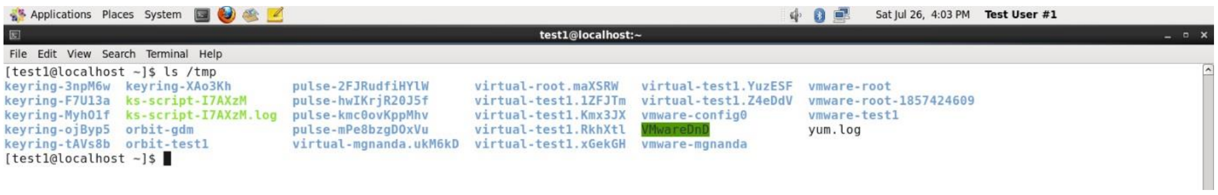
Temporary files (and directories) are meant to store data for a short time. Usually one arranges it so that these files disappear when the program using them terminates. While you can also use **touch** to create a temporary file, this may make it easy for hackers to gain access to your data.

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


Applications Places System
Sat Jul 26, 4:03 PM Test User #1
test1@localhost:~
File Edit View Search Terminal Help

```
[test1@localhost ~]$ ls /tmp
keyring-3npM6w  keyring-XAo3Kh        pulse-2FJRudfiHYlW      virtual-root.maXSRW   virtual-test1.YuzESF  vmware-root
keyring-F7U13a  ks-script-I7AXzM      pulse-hwIKrjR20J5f      virtual-test1.1ZFJTm  virtual-test1.Z4eDdV  vmware-root-1857424609
keyring-Myh01f  ks-script-I7AXzM.log  pulse-kmc0ovKppMhv      virtual-test1.Kmx3JX  vmware-config0        vmware-test1
keyring-ojByp5  orbit-gdm             pulse-mPe8bzgD0xVu      virtual-test1.RkhXtl  VMwareDnD             yum.log
keyring-tAVs8b  orbit-test1           virtual-mgnanda.ukM6kD  virtual-test1.xGekGH  vmware-mgnanda
[test1@localhost ~]$ █
```

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
```

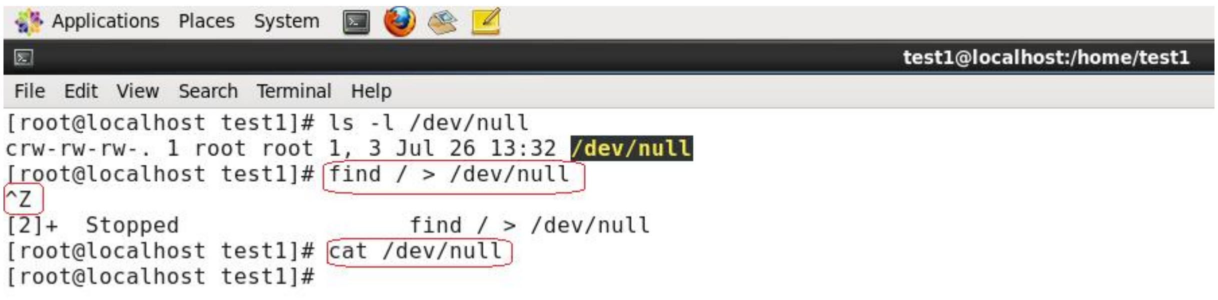
The password file will be overwritten by the temporary file contents.

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
```

Click the image to view an enlarged version.

Discarding Output with /dev/null


Applications Places System
test1@localhost:/home/test1
File Edit View Search Terminal Help

```
[root@localhost test1]# ls -l /dev/null
crw-rw-rw-. 1 root root 1, 3 Jul 26 13:32 /dev/null
[root@localhost test1]# find / > /dev/null
^Z
[2]+  Stopped                 find / > /dev/null
[root@localhost test1]# cat /dev/null
[root@localhost test1]#
```

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**.

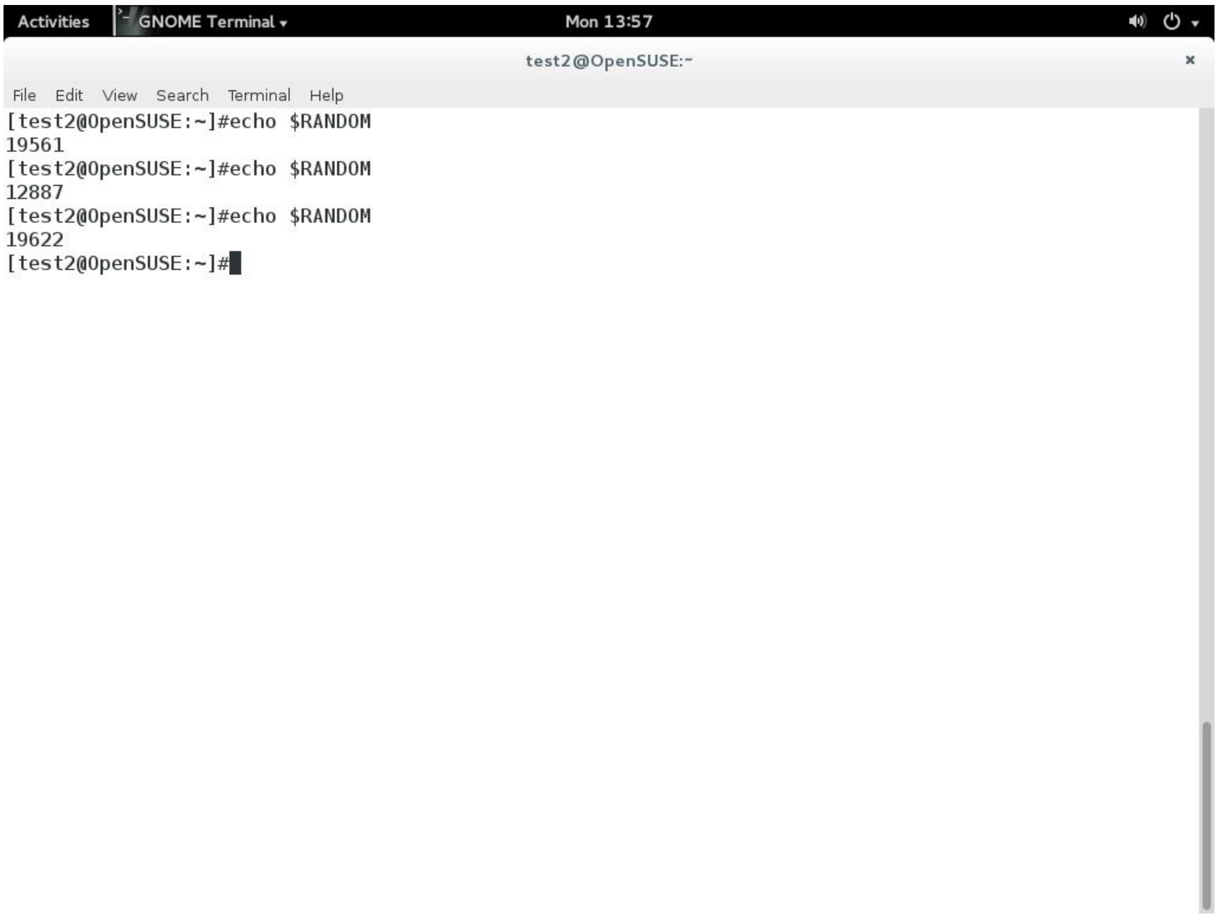
It discards all data that gets written to it and never returns a failure on write operations. Using the proper redirection operators, it can make the output disappear from commands that would normally generate output to **stdout** and/or **stderr**:

```
$ find / > /dev/null
```

In the above command, the entire standard output stream is ignored, but any errors will still appear on the console.

Click the image to view an enlarged version.

Random Numbers and Data



```
[test2@OpenSUSE:~]#echo $RANDOM
19561
[test2@OpenSUSE:~]#echo $RANDOM
12887
[test2@OpenSUSE:~]#echo $RANDOM
19622
[test2@OpenSUSE:~]#
```

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

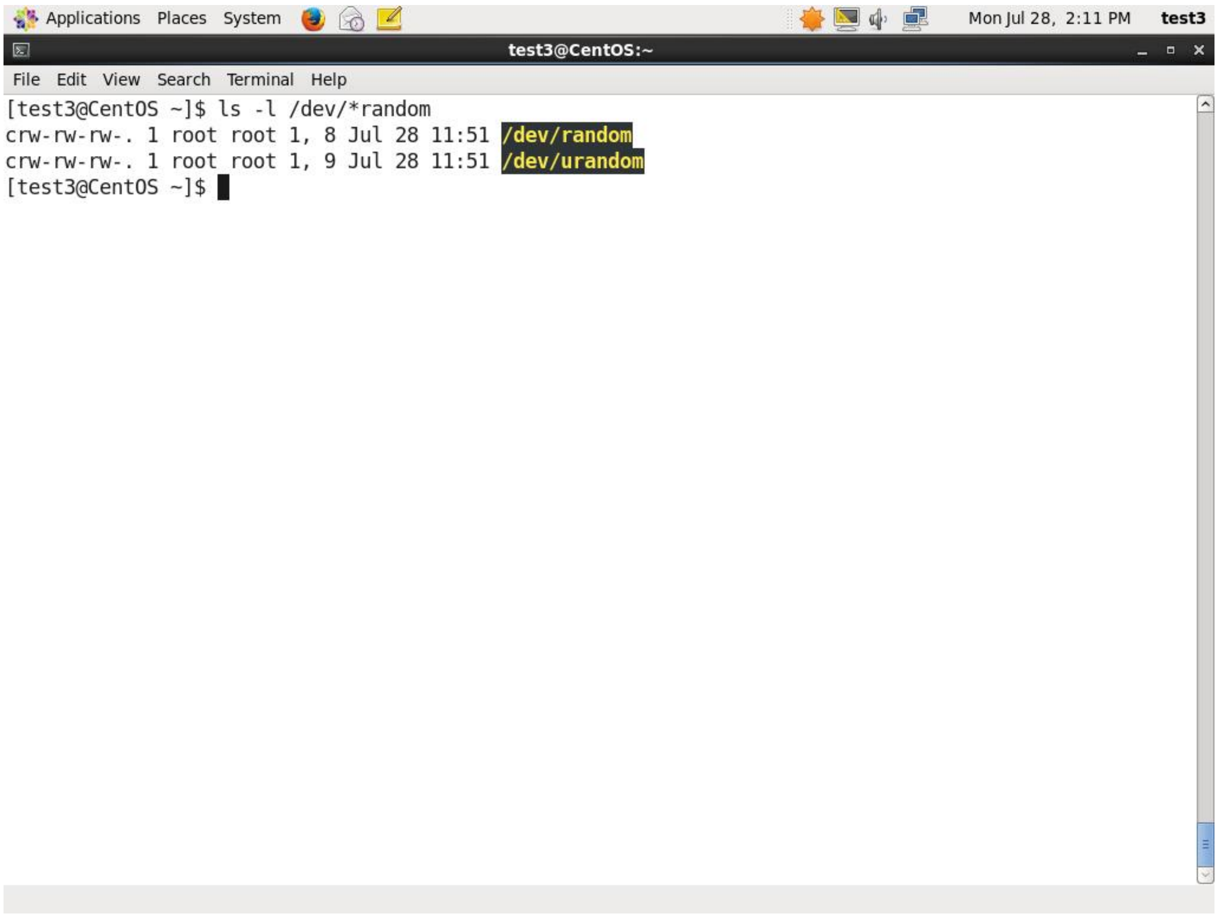
- Performing security-related tasks.
- Reinitializing storage devices.
- Erasing and/or obscuring existing data.
- Generating meaningless data to be used for tests.

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


Applications Places System
Mon Jul 28, 2:11 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

```
[test3@CentOS ~]$ ls -l /dev/*random
crw-rw-rw-. 1 root root 1, 8 Jul 28 11:51 /dev/random
crw-rw-rw-. 1 root root 1, 9 Jul 28 11:51 /dev/urandom
[test3@CentOS ~]$
```

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.

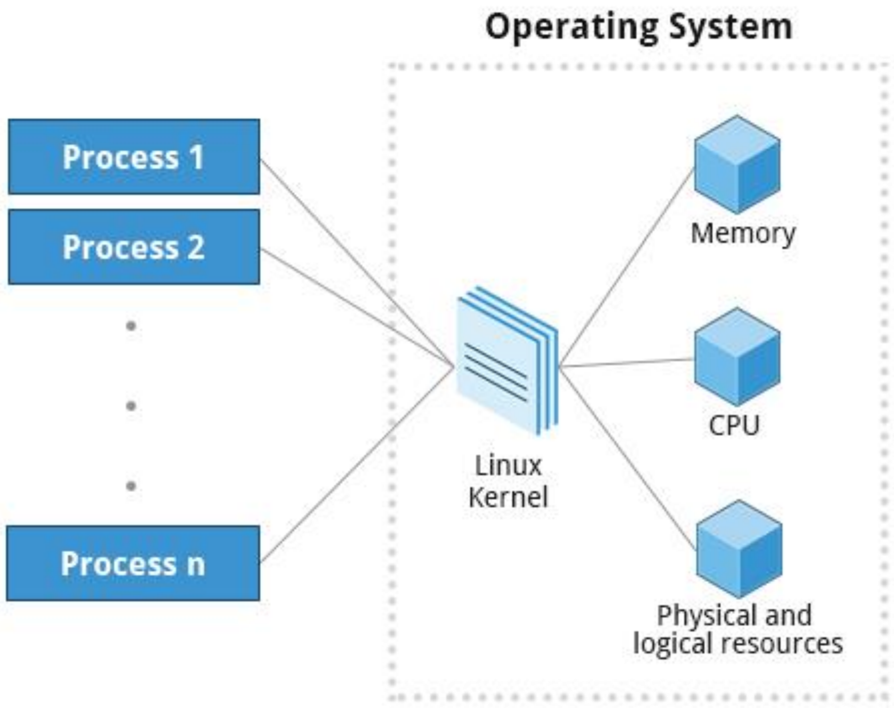
/dev/random is used where very high quality randomness is required, such as one-time pad or key generation, but it is relatively slow to provide vaules. **/dev/urandom** is faster and suitable (good enough) for most cryptographic purposes.

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.

Sekcja1

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

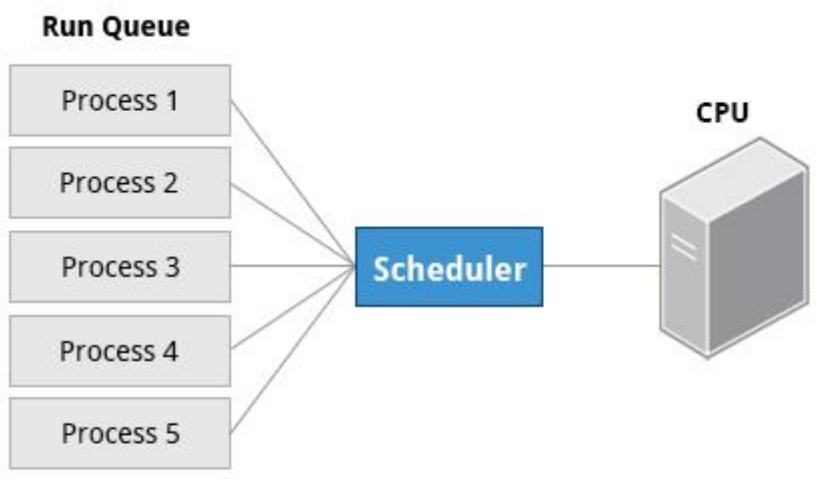
A terminal window (one kind of command shell), is a process that runs as long as needed. It allows users to execute programs and access resources in an interactive environment. You can also run programs in the **background**, which means they become **detached** from the shell.

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 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	**updatedb**

	tasks are queued and work on a **FIFO** (First In, First Out) basis.	
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**

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.

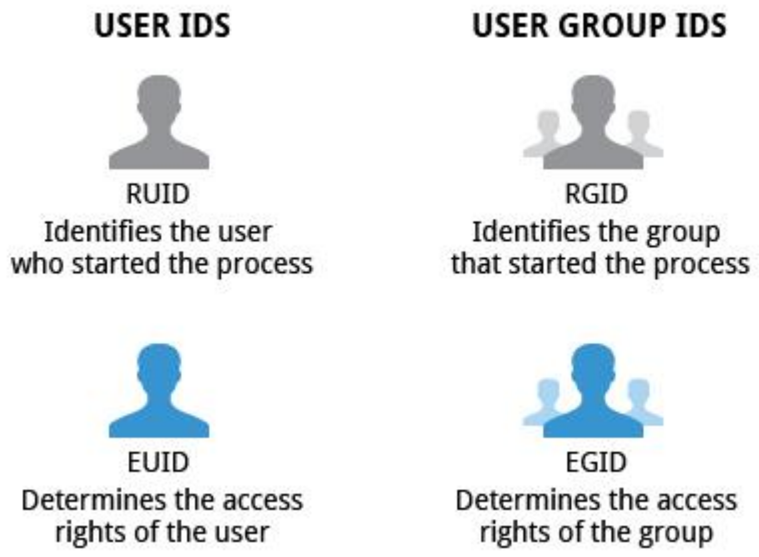
New PIDs are usually assigned in ascending order as processes are born. Thus PID 1 denotes the **init** process (initialization process), and succeeding processes are gradually assigned higher numbers.

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.


USER IDS
USER GROUP IDS
RUID
Identifies the user who started the process
RGID
Identifies the group that started the process
EUID
Determines the access rights of the user
EGID
Determines the access rights of the group

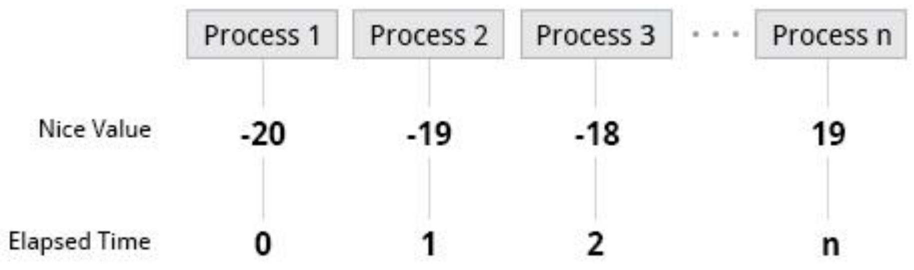
The user who determines the access rights for the users is identified by the Effective UID (**EUID**). The EUID may or may not be the same as the RUID.

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.

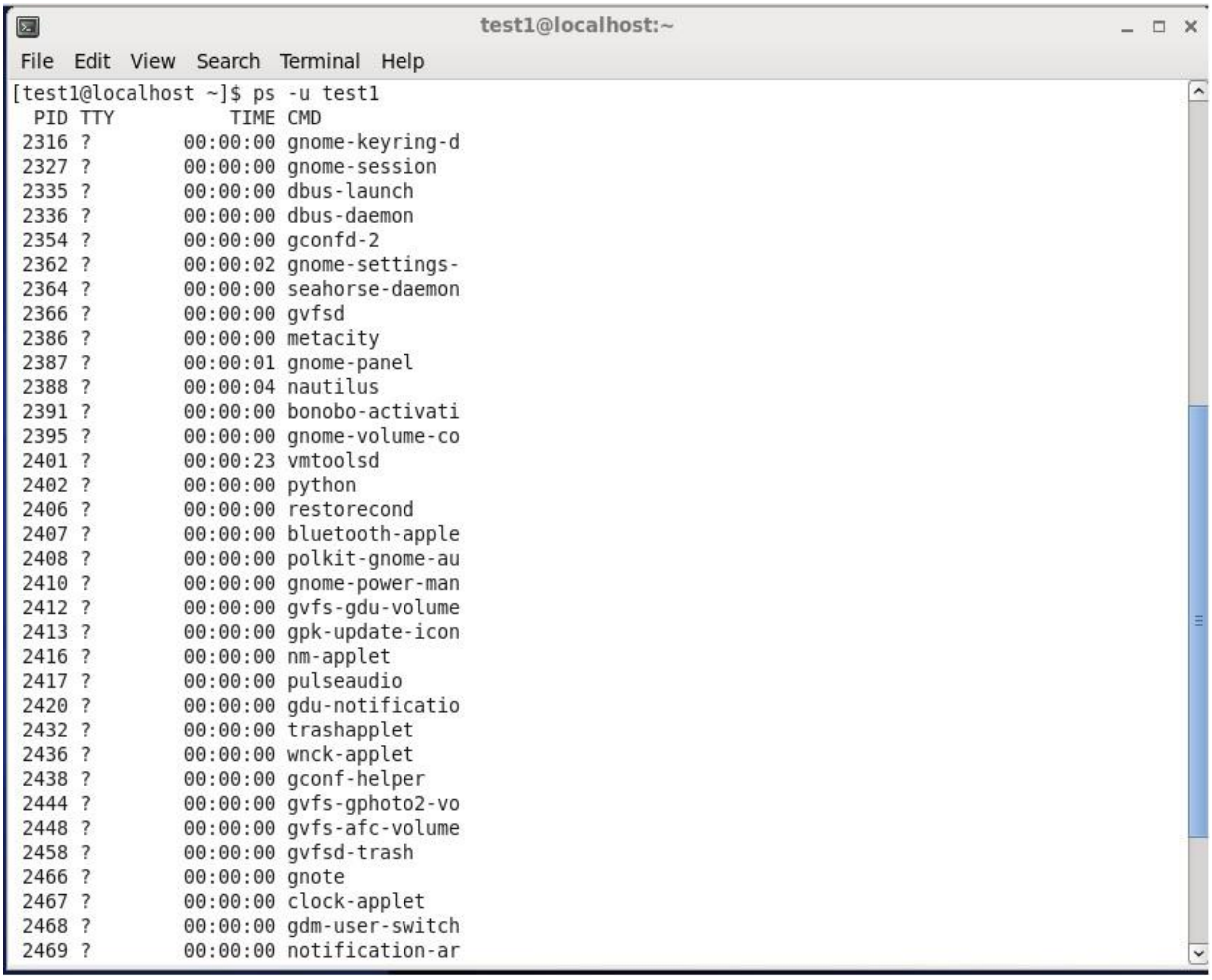


The priority for a process can be set by specifying a **nice value**, or **niceness**, for the process. The lower the nice value, the higher the priority. Low values are assigned to important processes, while high values are assigned to processes that can wait longer. A process with a high nice value 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.

Section2

The ps Command (System V Style)



```
[test1@localhost ~]$ ps -u test1
  PID TTY          TIME CMD
 2316 ?        00:00:00 gnome-keyring-d
 2327 ?        00:00:00 gnome-session
 2335 ?        00:00:00 dbus-launch
 2336 ?        00:00:00 dbus-daemon
 2354 ?        00:00:00 gconfd-2
 2362 ?        00:00:02 gnome-settings-
 2364 ?        00:00:00 seahorse-daemon
 2366 ?        00:00:00 gvfsd
 2386 ?        00:00:00 metacity
 2387 ?        00:00:01 gnome-panel
 2388 ?        00:00:04 nautilus
 2391 ?        00:00:00 bonobo-activati
 2395 ?        00:00:00 gnome-volume-co
 2401 ?        00:00:23 vmtoolsd
 2402 ?        00:00:00 python
 2406 ?        00:00:00 restorecond
 2407 ?        00:00:00 bluetooth-apple
 2408 ?        00:00:00 polkit-gnome-au
 2410 ?        00:00:00 gnome-power-man
 2412 ?        00:00:00 gvfs-gdu-volume
 2413 ?        00:00:00 gpk-update-icon
 2416 ?        00:00:00 nm-applet
 2417 ?        00:00:00 pulseaudio
 2420 ?        00:00:00 gdu-notificatio
 2432 ?        00:00:00 trashapplet
 2436 ?        00:00:00 wnck-applet
 2438 ?        00:00:00 gconf-helper
 2444 ?        00:00:00 gvfs-gphoto2-vo
 2448 ?        00:00:00 gvfs-afc-volume
 2458 ?        00:00:00 gvfsd-trash
 2466 ?        00:00:00 gnote
 2467 ?        00:00:00 clock-applet
 2468 ?        00:00:00 gdm-user-switch
 2469 ?        00:00:00 notification-ar
```

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 tables shows sample output of **ps** with the `aux` and `axo` qualifiers.

Command	Output

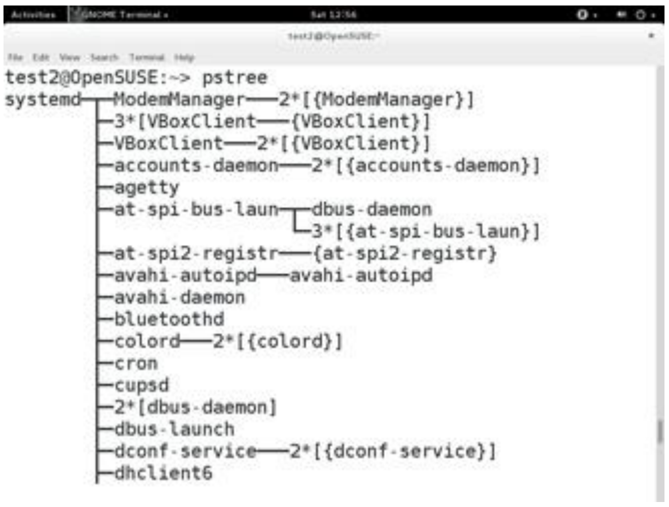
`ps aux`	`USER PID %CPU %MEM VSZ   RSS  TTY STAT START TIME COMMAND`
	`root 1   0.0  0.0  19356 1292 ?   Ss   Feb27 0:08`

	`/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]`
	
	`. . .`

Command	Output

`ps aux`
`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



```
test2@OpenSUSE:~> pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─3*[VBoxClient───{VBoxClient}]
        ├─VBoxClient───2*[{VBoxClient}]
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─agetty
        ├─at-spi-bus-laun─┬─dbus-daemon
        │                 └─3*[{at-spi-bus-laun}]
        ├─at-spi2-registr───{at-spi2-registr}
        ├─avahi-autoipd───avahi-autoipd
        ├─avahi-daemon
        ├─bluetoothd
        ├─colord───2*[{colord}]
        ├─cron
        ├─cupsd
        ├─2*[dbus-daemon]
        ├─dbus-launch
        ├─dconf-service───2*[{dconf-service}]
        ├─dhclient6
```

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

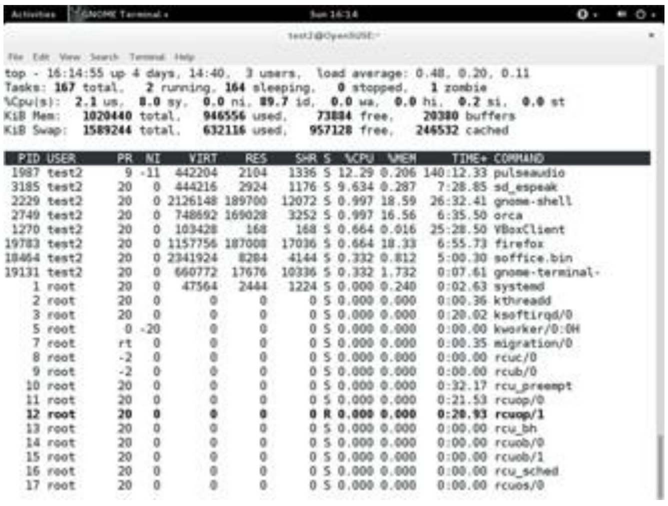
pstree displays the processes running on the system in the form of a **tree diagram** showing the relationship between a process and its parent process and any other processes that it created. Repeated entries of a process are not displayed, and threads are displayed in curly braces.

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

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**.)

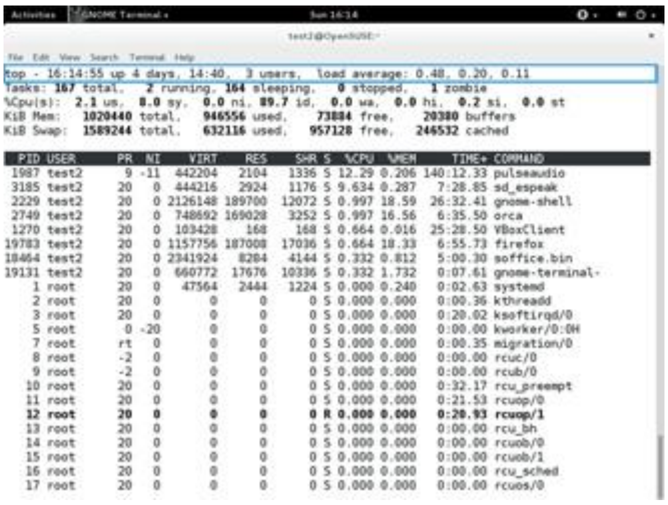


```
top - 16:14:55 up 4 days, 14:40,  3 users,  load average: 0.48, 0.20, 0.11
Tasks: 167 total,   2 running, 164 sleeping,   0 stopped,   1 zombie
%Cpu(s):  2.1 us,  8.0 sy,  0.0 ni, 89.7 id,  0.0 wa,  0.0 hi,  0.2 si,  0.0 st
KiB Mem:   1020440 total,   946556 used,    73884 free,    20380 buffers
KiB Swap:  1589244 total,   632116 used,   957128 free,   246532 cached

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1987 test2      9 -11  442204   2104   1336 S 12.29 0.206 140:12.33 pulseaudio
 3185 test2     20   0  444216   2924   1176 S 9.634 0.287   7:28.85 sd_espeak
 2229 test2     20   0 2126148 189700  12072 S 0.997 18.59  26:32.41 gnome-shell
 2749 test2     20   0  748692 169028   3252 S 0.997 16.56   6:35.50 orca
 1270 test2     20   0  103428    168    168 S 0.664 0.016  25:28.50 VBoxClient
19783 test2     20   0 1157756 187008  17036 S 0.664 18.33   6:55.73 firefox
18464 test2     20   0 2341924   8284   4144 S 0.332 0.812   5:00.30 soffice.bin
19131 test2     20   0  660772  17676  10336 S 0.332 1.732   0:07.61 gnome-terminal-
    1 root      20   0   47564   2444   1224 S 0.000 0.240   0:02.63 systemd
    2 root      20   0       0      0      0 S 0.000 0.000   0:00.36 kthreadd
    3 root      20   0       0      0      0 S 0.000 0.000   0:20.02 ksoftirqd/0
    5 root       0 -20       0      0      0 S 0.000 0.000   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S 0.000 0.000   0:00.35 migration/0
    8 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcuc/0
    9 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcub/0
   10 root      20   0       0      0      0 S 0.000 0.000   0:32.17 rcu_preempt
   11 root      20   0       0      0      0 S 0.000 0.000   0:21.53 rcuop/0
   12 root      20   0       0      0      0 R 0.000 0.000   0:20.93 rcuop/1
   13 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_bh
   14 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/0
   15 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/1
   16 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_sched
   17 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuos/0
```

First Line of the top Output

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



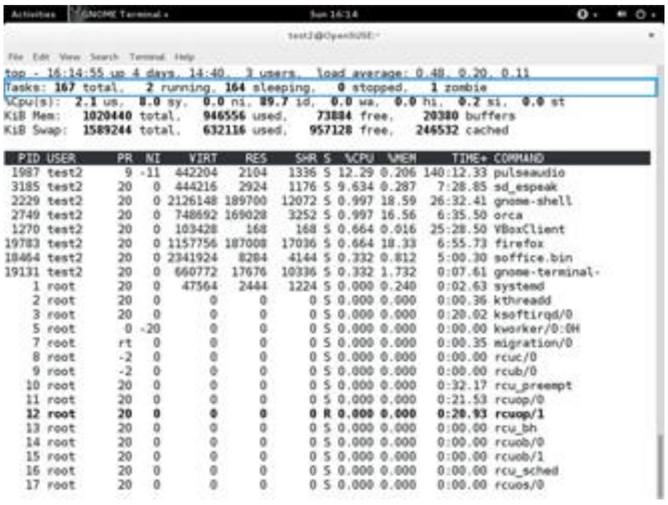
```
top - 16:14:55 up 4 days, 14:40,  3 users,  load average: 0.48, 0.20, 0.11
Tasks: 167 total,   2 running, 164 sleeping,   0 stopped,   1 zombie
%Cpu(s):  2.1 us,  8.0 sy,  0.0 ni, 89.7 id,  0.0 wa,  0.0 hi,  0.2 si,  0.0 st
KiB Mem:   1020440 total,   946556 used,    73884 free,    20380 buffers
KiB Swap:  1589244 total,   632116 used,   957128 free,   246532 cached

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1987 test2      9 -11  442204   2104   1336 S 12.29 0.206 140:12.33 pulseaudio
 3185 test2     20   0  444216   2924   1176 S 9.634 0.287   7:28.85 sd_espeak
 2229 test2     20   0 2126148 189700  12072 S 0.997 18.59  26:32.41 gnome-shell
 2749 test2     20   0  748692 169028   3252 S 0.997 16.56   6:35.50 orca
 1270 test2     20   0  103428    168    168 S 0.664 0.016  25:28.50 VBoxClient
19783 test2     20   0 1157756 187008  17036 S 0.664 18.33   6:55.73 firefox
18464 test2     20   0 2341924   8284   4144 S 0.332 0.812   5:00.30 soffice.bin
19131 test2     20   0  660772  17676  10336 S 0.332 1.732   0:07.61 gnome-terminal-
    1 root      20   0   47564   2444   1224 S 0.000 0.240   0:02.63 systemd
    2 root      20   0       0      0      0 S 0.000 0.000   0:00.36 kthreadd
    3 root      20   0       0      0      0 S 0.000 0.000   0:20.02 ksoftirqd/0
    5 root       0 -20       0      0      0 S 0.000 0.000   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S 0.000 0.000   0:00.35 migration/0
    8 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcuc/0
    9 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcub/0
   10 root      20   0       0      0      0 S 0.000 0.000   0:32.17 rcu_preempt
   11 root      20   0       0      0      0 S 0.000 0.000   0:21.53 rcuop/0
   12 root      20   0       0      0      0 R 0.000 0.000   0:20.93 rcuop/1
   13 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_bh
   14 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/0
   15 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/1
   16 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_sched
   17 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuos/0
```

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

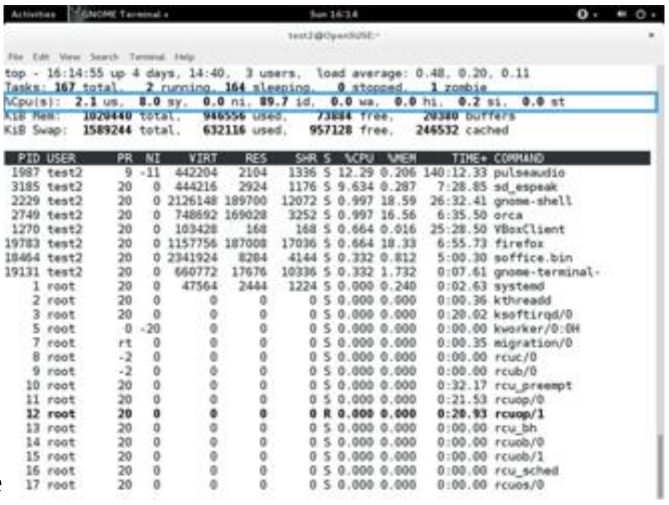
The second line of the **top** output displays the total number of processes, the number of running, sleeping, stopped and zombie processes. Comparing the number of running processes with the load average helps determine if the system has reached its capacity or perhaps a particular user is running too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

Click the image to view an enlarged version.



```
top - 16:14:55 up 4 days, 14:40,  3 users,  load average: 0.48, 0.20, 0.11
Tasks: 167 total,   2 running, 164 sleeping,   0 stopped,   1 zombie
%Cpu(s):  2.1 us,  8.0 sy,  0.0 ni, 89.7 id,  0.0 wa,  0.0 hi,  0.2 si,  0.0 st
KiB Mem:   1020440 total,   946556 used,    73884 free,    20380 buffers
KiB Swap:  1589244 total,   632116 used,   957128 free,   246532 cached

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1987 test2      9 -11  442204   2104   1336 S 12.29 0.206 140:12.33 pulseaudio
 3185 test2     20   0  444216   2924   1176 S 9.634 0.287   7:28.85 sd_espeak
 2229 test2     20   0 2126148 189700  12072 S 0.997 18.59  26:32.41 gnome-shell
 2749 test2     20   0  748692 169028   3252 S 0.997 16.56   6:35.50 orca
 1270 test2     20   0  103428    168    168 S 0.664 0.016  25:28.50 VBoxClient
19783 test2     20   0 1157756 187008  17036 S 0.664 18.33   6:55.73 firefox
18464 test2     20   0 2341924   8284   4144 S 0.332 0.812   5:00.30 soffice.bin
19131 test2     20   0  660772  17676  10336 S 0.332 1.732   0:07.61 gnome-terminal-
    1 root      20   0   47564   2444   1224 S 0.000 0.240   0:02.63 systemd
    2 root      20   0       0      0      0 S 0.000 0.000   0:00.36 kthreadd
    3 root      20   0       0      0      0 S 0.000 0.000   0:20.02 ksoftirqd/0
    5 root       0 -20       0      0      0 S 0.000 0.000   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S 0.000 0.000   0:00.35 migration/0
    8 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcuc/0
    9 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcub/0
   10 root      20   0       0      0      0 S 0.000 0.000   0:32.17 rcu_preempt
   11 root      20   0       0      0      0 S 0.000 0.000   0:21.53 rcuop/0
   12 root      20   0       0      0      0 R 0.000 0.000   0:20.93 rcuop/1
   13 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_bh
   14 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/0
   15 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/1
   16 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_sched
   17 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuos/0
```

Third Line of the top Output



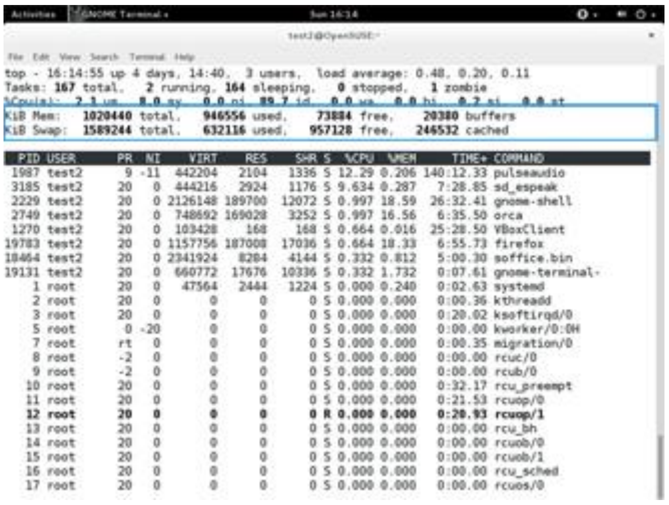
```
top - 16:14:55 up 4 days, 14:40,  3 users,  load average: 0.48, 0.20, 0.11
Tasks: 167 total,   2 running, 164 sleeping,   0 stopped,   1 zombie
%Cpu(s):  2.1 us,  8.0 sy,  0.0 ni, 89.7 id,  0.0 wa,  0.0 hi,  0.2 si,  0.0 st
KiB Mem:   1020440 total,   946556 used,    73884 free,    20380 buffers
KiB Swap:  1589244 total,   632116 used,   957128 free,   246532 cached

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1987 test2      9 -11  442204   2104   1336 S 12.29 0.206 140:12.33 pulseaudio
 3185 test2     20   0  444216   2924   1176 S 9.634 0.287   7:28.85 sd_espeak
 2229 test2     20   0 2126148 189700  12072 S 0.997 18.59  26:32.41 gnome-shell
 2749 test2     20   0  748692 169028   3252 S 0.997 16.56   6:35.50 orca
 1270 test2     20   0  103428    168    168 S 0.664 0.016  25:28.50 VBoxClient
19783 test2     20   0 1157756 187008  17036 S 0.664 18.33   6:55.73 firefox
18464 test2     20   0 2341924   8284   4144 S 0.332 0.812   5:00.30 soffice.bin
19131 test2     20   0  660772  17676  10336 S 0.332 1.732   0:07.61 gnome-terminal-
    1 root      20   0   47564   2444   1224 S 0.000 0.240   0:02.63 systemd
    2 root      20   0       0      0      0 S 0.000 0.000   0:00.36 kthreadd
    3 root      20   0       0      0      0 S 0.000 0.000   0:20.02 ksoftirqd/0
    5 root       0 -20       0      0      0 S 0.000 0.000   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S 0.000 0.000   0:00.35 migration/0
    8 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcuc/0
    9 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcub/0
   10 root      20   0       0      0      0 S 0.000 0.000   0:32.17 rcu_preempt
   11 root      20   0       0      0      0 S 0.000 0.000   0:21.53 rcuop/0
   12 root      20   0       0      0      0 R 0.000 0.000   0:20.93 rcuop/1
   13 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_bh
   14 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/0
   15 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/1
   16 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_sched
   17 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuos/0
```

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



```
top - 16:14:55 up 4 days, 14:40,  3 users,  load average: 0.48, 0.20, 0.11
Tasks: 167 total,   2 running, 164 sleeping,   0 stopped,   1 zombie
%Cpu(s):  2.1 us,  8.0 sy,  0.0 ni, 89.7 id,  0.0 wa,  0.0 hi,  0.2 si,  0.0 st
KiB Mem:   1020440 total,   946556 used,    73884 free,    20380 buffers
KiB Swap:  1589244 total,   632116 used,   957128 free,   246532 cached

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1987 test2      9 -11  442204   2104   1336 S 12.29 0.206 140:12.33 pulseaudio
 3185 test2     20   0  444216   2924   1176 S 9.634 0.287   7:28.85 sd_espeak
 2229 test2     20   0 2126148 189700  12072 S 0.997 18.59  26:32.41 gnome-shell
 2749 test2     20   0  748692 169028   3252 S 0.997 16.56   6:35.50 orca
 1270 test2     20   0  103428    168    168 S 0.664 0.016  25:28.50 VBoxClient
19783 test2     20   0 1157756 187008  17036 S 0.664 18.33   6:55.73 firefox
18464 test2     20   0 2341924   8284   4144 S 0.332 0.812   5:00.30 soffice.bin
19131 test2     20   0  660772  17676  10336 S 0.332 1.732   0:07.61 gnome-terminal-
    1 root      20   0   47564   2444   1224 S 0.000 0.240   0:02.63 systemd
    2 root      20   0       0      0      0 S 0.000 0.000   0:00.36 kthreadd
    3 root      20   0       0      0      0 S 0.000 0.000   0:20.02 ksoftirqd/0
    5 root       0 -20       0      0      0 S 0.000 0.000   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S 0.000 0.000   0:00.35 migration/0
    8 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcuc/0
    9 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcub/0
   10 root      20   0       0      0      0 S 0.000 0.000   0:32.17 rcu_preempt
   11 root      20   0       0      0      0 S 0.000 0.000   0:21.53 rcuop/0
   12 root      20   0       0      0      0 R 0.000 0.000   0:20.93 rcuop/1
   13 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_bh
   14 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/0
   15 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/1
   16 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_sched
   17 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuos/0
```

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.

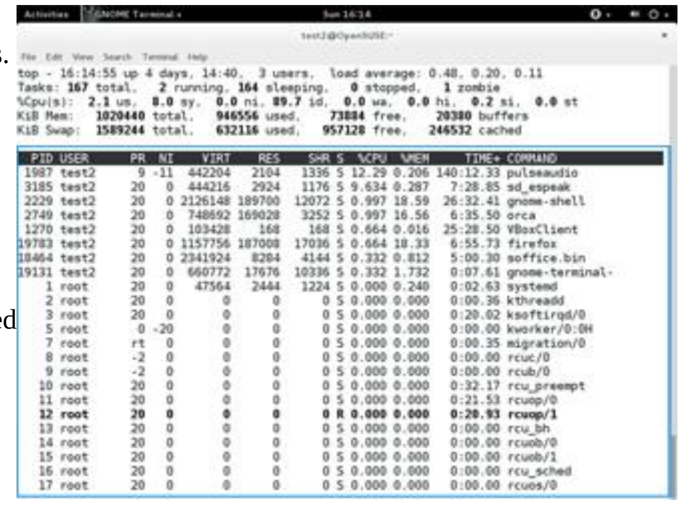
If the system starts using swap often, you can add more swap space. However, adding more physical memory should also be considered.

Click the image to view an enlarged version.

Process List of the top Output

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)



```
top - 16:14:55 up 4 days, 14:40,  3 users,  load average: 0.48, 0.20, 0.11
Tasks: 167 total,   2 running, 164 sleeping,   0 stopped,   1 zombie
%Cpu(s):  2.1 us,  8.0 sy,  0.0 ni, 89.7 id,  0.0 wa,  0.0 hi,  0.2 si,  0.0 st
KiB Mem:   1020440 total,   946556 used,    73884 free,    20380 buffers
KiB Swap:  1589244 total,   632116 used,   957128 free,   246532 cached

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1987 test2      9 -11  442204   2104   1336 S 12.29 0.206 140:12.33 pulseaudio
 3185 test2     20   0  444216   2924   1176 S 9.634 0.287   7:28.85 sd_espeak
 2229 test2     20   0 2126148 189700  12072 S 0.997 18.59  26:32.41 gnome-shell
 2749 test2     20   0  748692 169028   3252 S 0.997 16.56   6:35.50 orca
 1270 test2     20   0  103428    168    168 S 0.664 0.016  25:28.50 VBoxClient
19783 test2     20   0 1157756 187008  17036 S 0.664 18.33   6:55.73 firefox
18464 test2     20   0 2341924   8284   4144 S 0.332 0.812   5:00.30 soffice.bin
19131 test2     20   0  660772  17676  10336 S 0.332 1.732   0:07.61 gnome-terminal-
    1 root      20   0   47564   2444   1224 S 0.000 0.240   0:02.63 systemd
    2 root      20   0       0      0      0 S 0.000 0.000   0:00.36 kthreadd
    3 root      20   0       0      0      0 S 0.000 0.000   0:20.02 ksoftirqd/0
    5 root       0 -20       0      0      0 S 0.000 0.000   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S 0.000 0.000   0:00.35 migration/0
    8 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcuc/0
    9 root      -2   0       0      0      0 S 0.000 0.000   0:00.00 rcub/0
   10 root      20   0       0      0      0 S 0.000 0.000   0:32.17 rcu_preempt
   11 root      20   0       0      0      0 S 0.000 0.000   0:21.53 rcuop/0
   12 root      20   0       0      0      0 R 0.000 0.000   0:20.93 rcuop/1
   13 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_bh
   14 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/0
   15 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuob/1
   16 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_sched
   17 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcuos/0
```

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

Section3

Load Averages

Load average is the average of the **load number** for a given period 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**

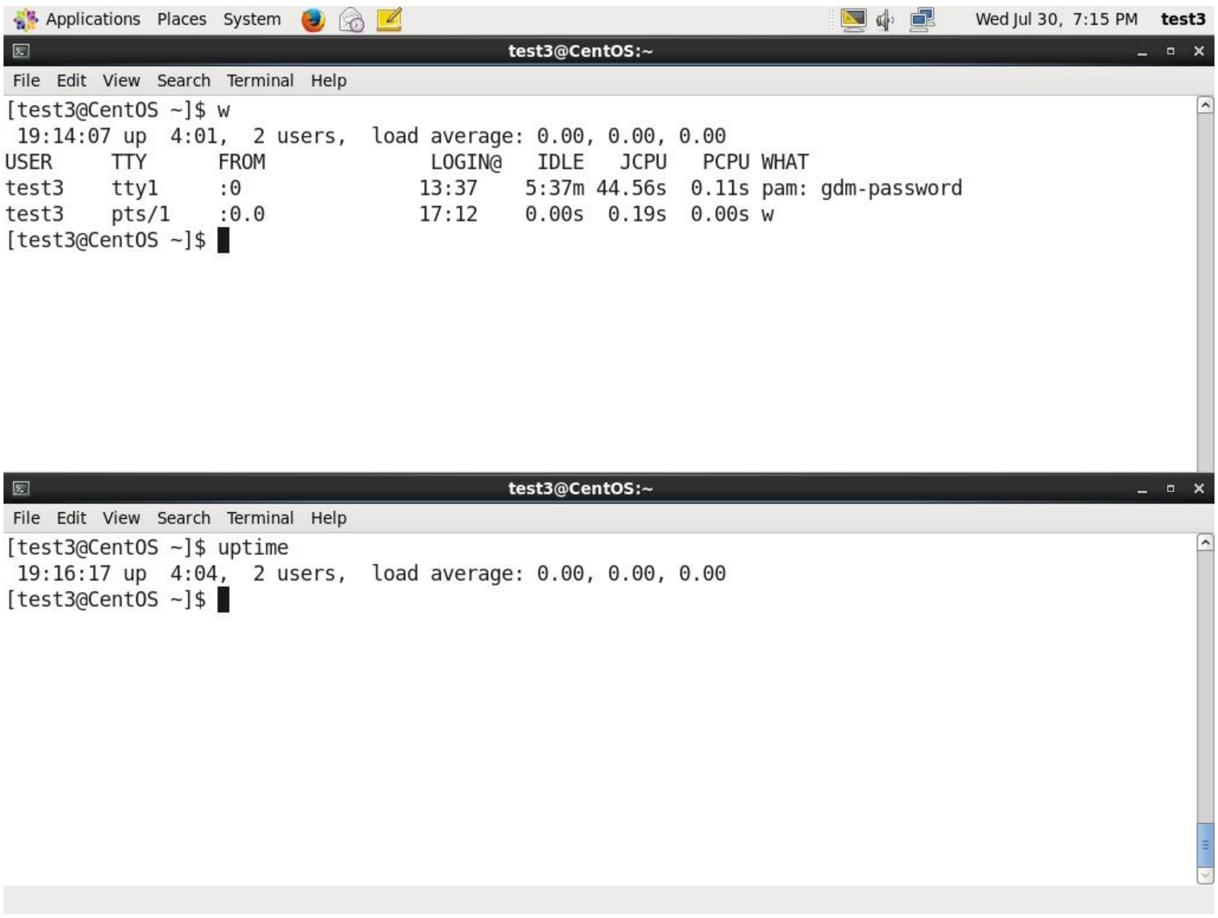
Interpreting Load Averages

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.


Applications Places System
Wed Jul 30, 7:15 PM test3
test3@CentOS:~
File Edit View Search Terminal Help

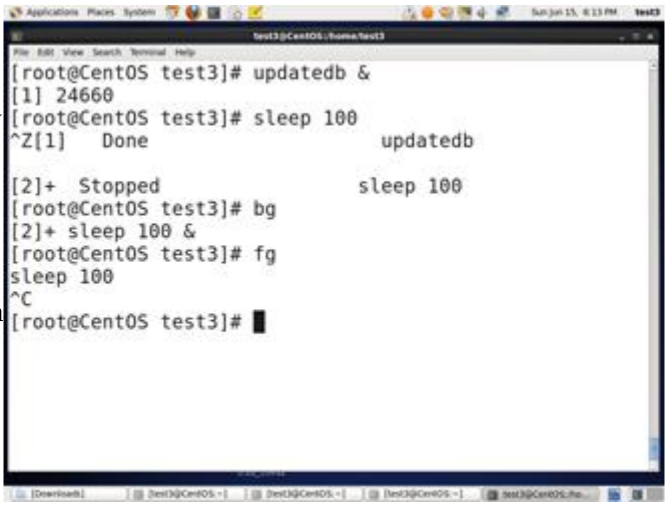
```
[test3@CentOS ~]$ w
 19:14:07 up  4:01,  2 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
test3    tty1     :0               13:37    5:37m 44.56s  0.11s pam: gdm-password
test3    pts/1    :0.0             17:12    0.00s  0.19s  0.00s w
[test3@CentOS ~]$
```

test3@CentOS:~
File Edit View Search Terminal Help

```
[test3@CentOS ~]$ uptime
 19:16:17 up  4:04,  2 users,  load average: 0.00, 0.00, 0.00
[test3@CentOS ~]$
```

Background and Foreground Processes

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.



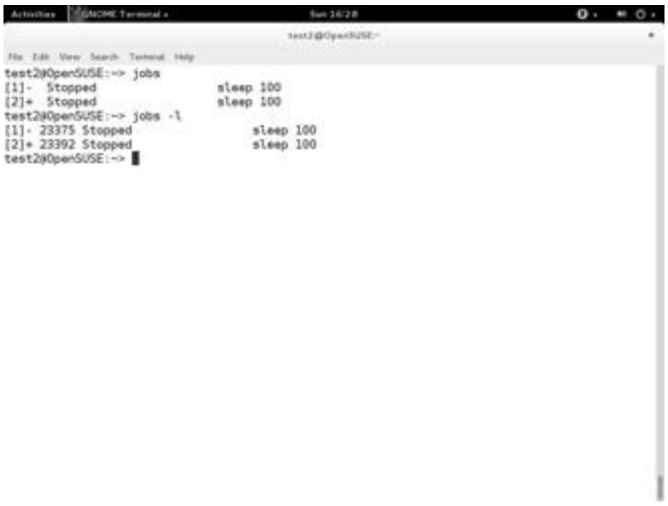
```
[root@CentOS test3]# updatedb &
[1] 24660
[root@CentOS test3]# sleep 100
^Z[1]   Done                    updatedb

[2]+  Stopped                 sleep 100
[root@CentOS test3]# bg
[2]+ sleep 100 &
[root@CentOS test3]# fg
sleep 100
^C
[root@CentOS test3]# 
```

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

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



```
test2@OpenSUSE:~> jobs
[1]-  Stopped                 sleep 100
[2]+  Stopped                 sleep 100
test2@OpenSUSE:~> jobs -l
[1]- 23375 Stopped                 sleep 100
[2]+ 23392 Stopped                 sleep 100
test2@OpenSUSE:~> 
```

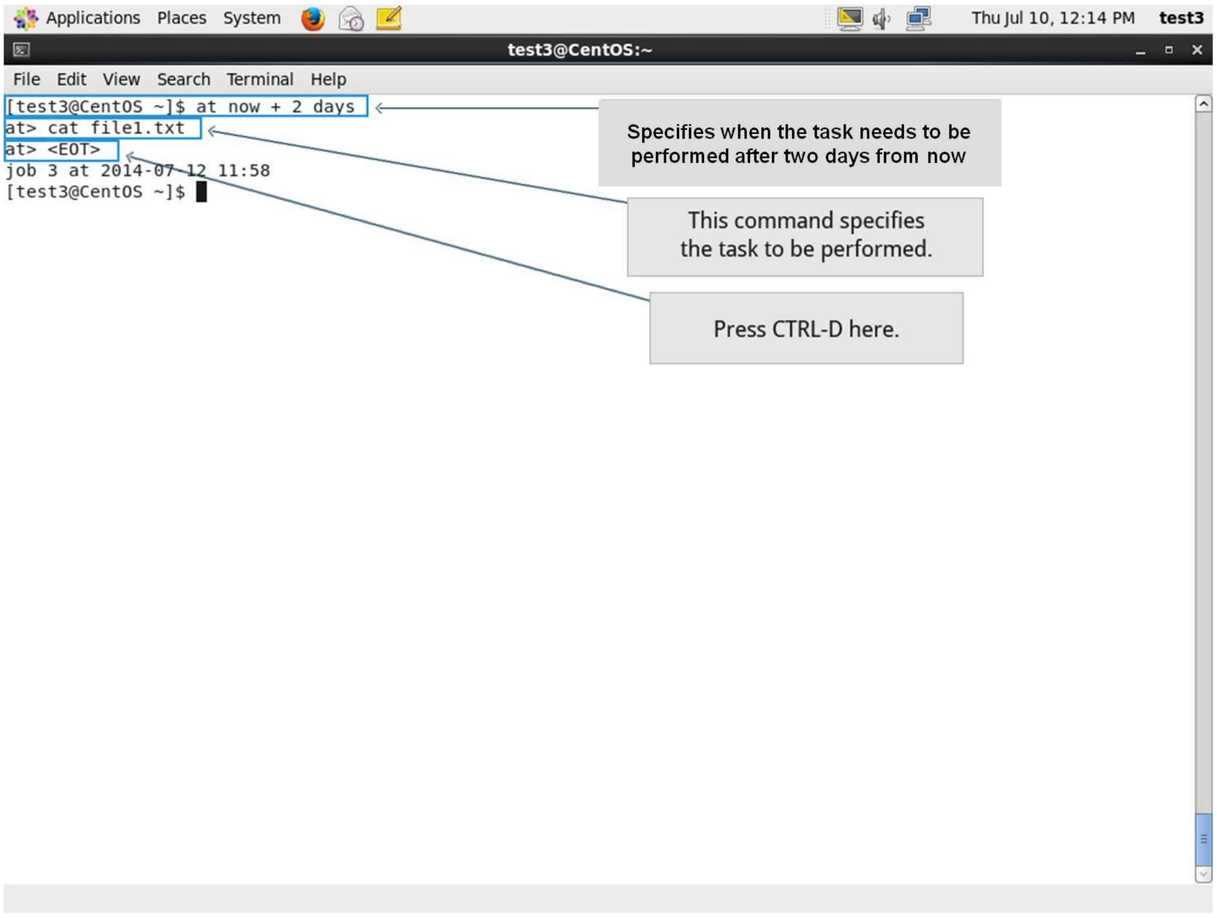
`jobs -l` provides a the same information as `jobs` including the PID of the background jobs.

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.

Click the image to view an enlarged version.

Section4

Scheduling Future Processes using at


Applications Places System
Thu Jul 10, 12:14 PM test3
test3@CentOS:~
File Edit View Search Terminal Help
Specifies when the task needs to be performed after two days from now
This command specifies the task to be performed.
Press CTRL-D here.

```
[test3@CentOS ~]$ at now + 2 days
at> cat file1.txt
at> <EOT>
job 3 at 2014-07-12 11:58
[test3@CentOS ~]$ 
```

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:

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:

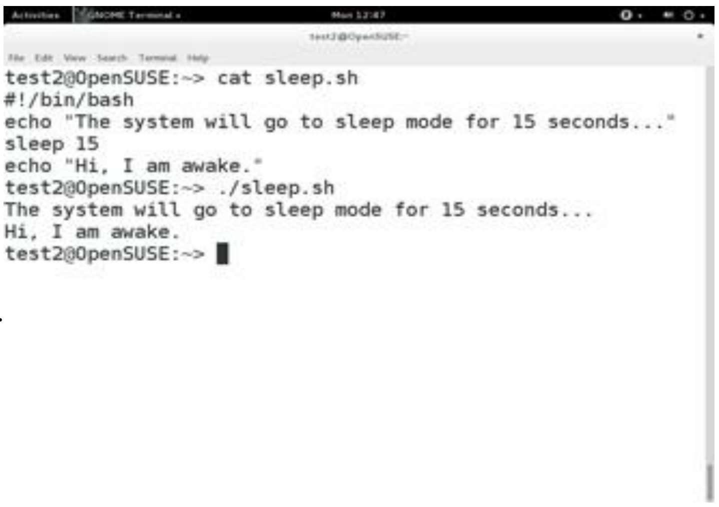
Field	Description	Values
MIN	Minutes	0 to 59
HOUR	Hour field	0 to 23
DOM	Day of Month	1-31
MON	Month field	1-12
DOW	Day Of Week	0-6 (0 = Sunday)

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



```
test2@OpenSUSE:~> cat sleep.sh
#!/bin/bash
echo "The system will go to sleep mode for 15 seconds..."
sleep 15
echo "Hi, I am awake."
test2@OpenSUSE:~> ./sleep.sh
The system will go to sleep mode for 15 seconds...
Hi, I am awake.
test2@OpenSUSE:~> █
```

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.