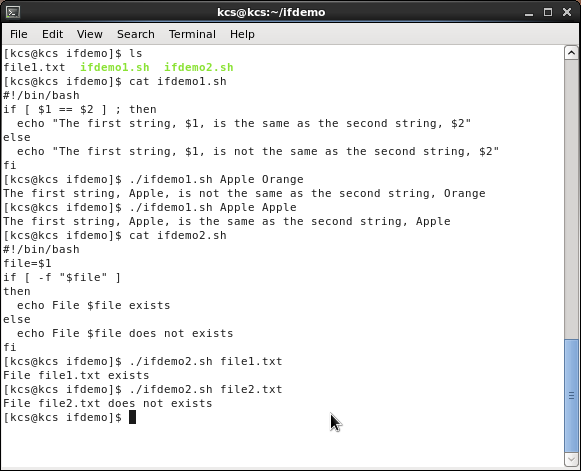
**String Manipulation**

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

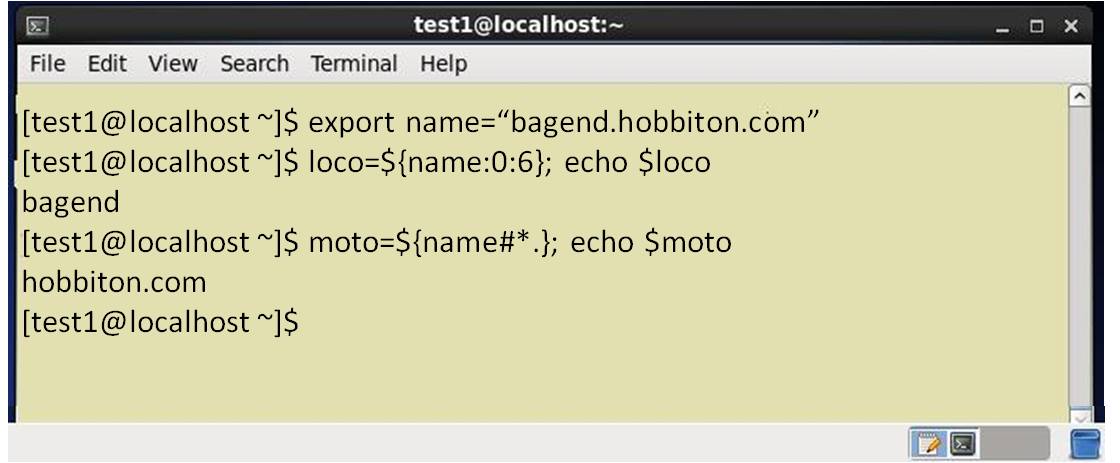
A **string variable** contains a sequence of text characters. It can include letters, numbers, symbols and punctuation marks. Some examples: abcde, 123, abcde 123, abcde-123, &acbde=%123

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. |



***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.*

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

***Boolean Expressions***

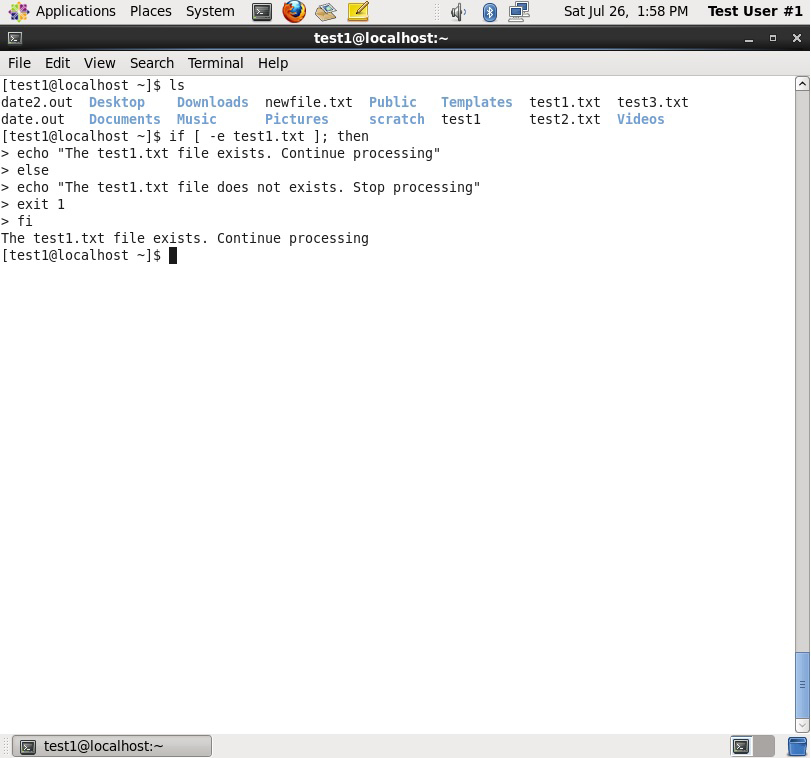
***Boolean*** *expressions evaluate to either* ***TRUE*** *or* ***FALSE****, and results are obtained using the various Boolean operators listed in the table.*

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

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

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

***Tests in Boolean Expressions***



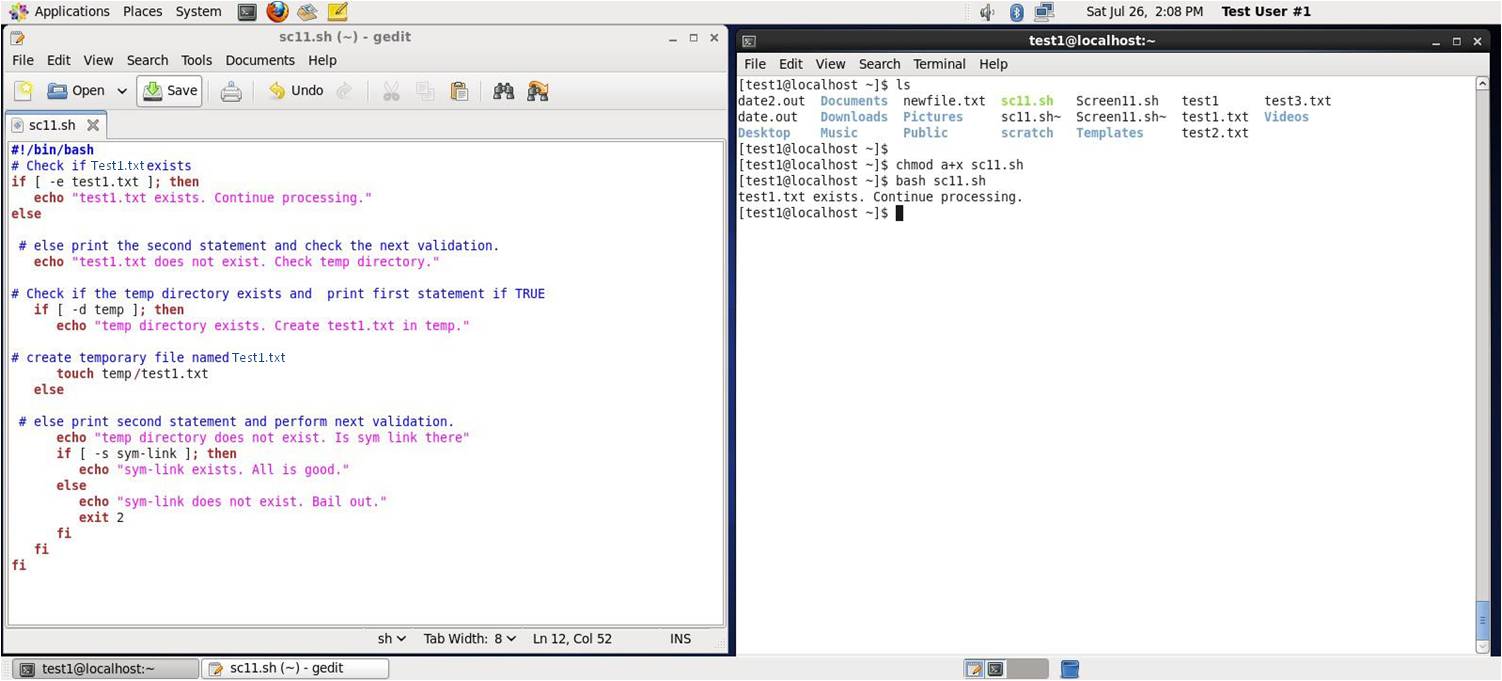
*Boolean expressions return either* ***TRUE*** *or* ***FALSE****. We can use such expressions when working with multiple data types including strings or numbers as well as with files. For example, to check if a file exists, use the following conditional test:*

*[ -e <filename> ]*

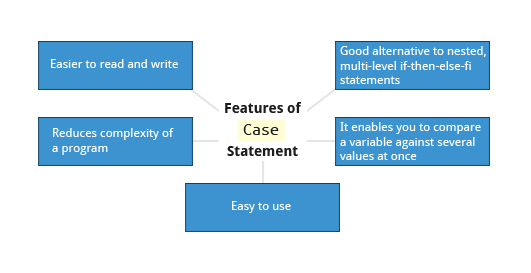
*Similarly, to check if the value of number1 is greater than the value of number2, use the following conditional test:*

*[ $number1 -gt $number2 ]*

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



***The case Statement***

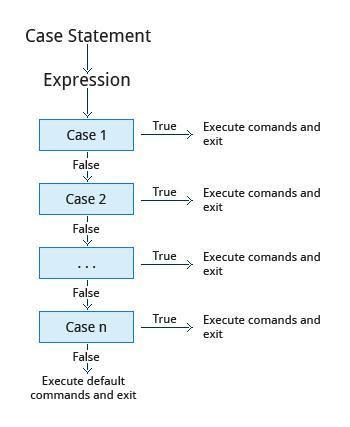


*The case statement is used in scenarios where the actual value of a variable can lead to different execution paths. case statements are often used to handle command-line options.*

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

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

***Structure of the case Statement***



*Here is the basic structure of the case statement:*

*case expression in*

*pattern1) execute commands;;*

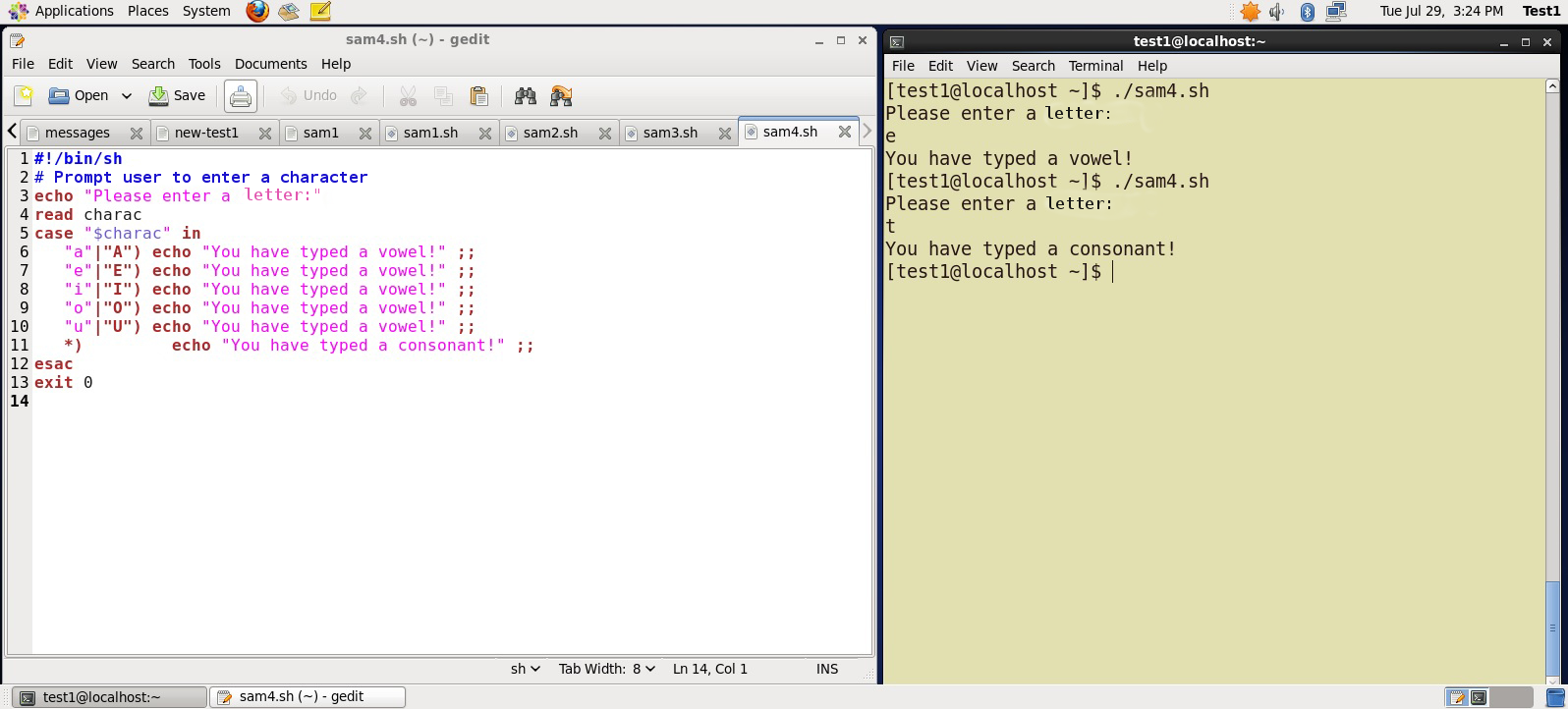
*pattern2) execute commands;;*

*pattern3) execute commands;;*

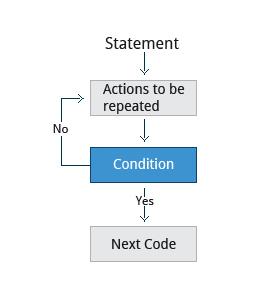
*pattern4) execute commands;;*

*\* ) execute some default commands or nothing ;;*

*esac*



**Looping Constructs**



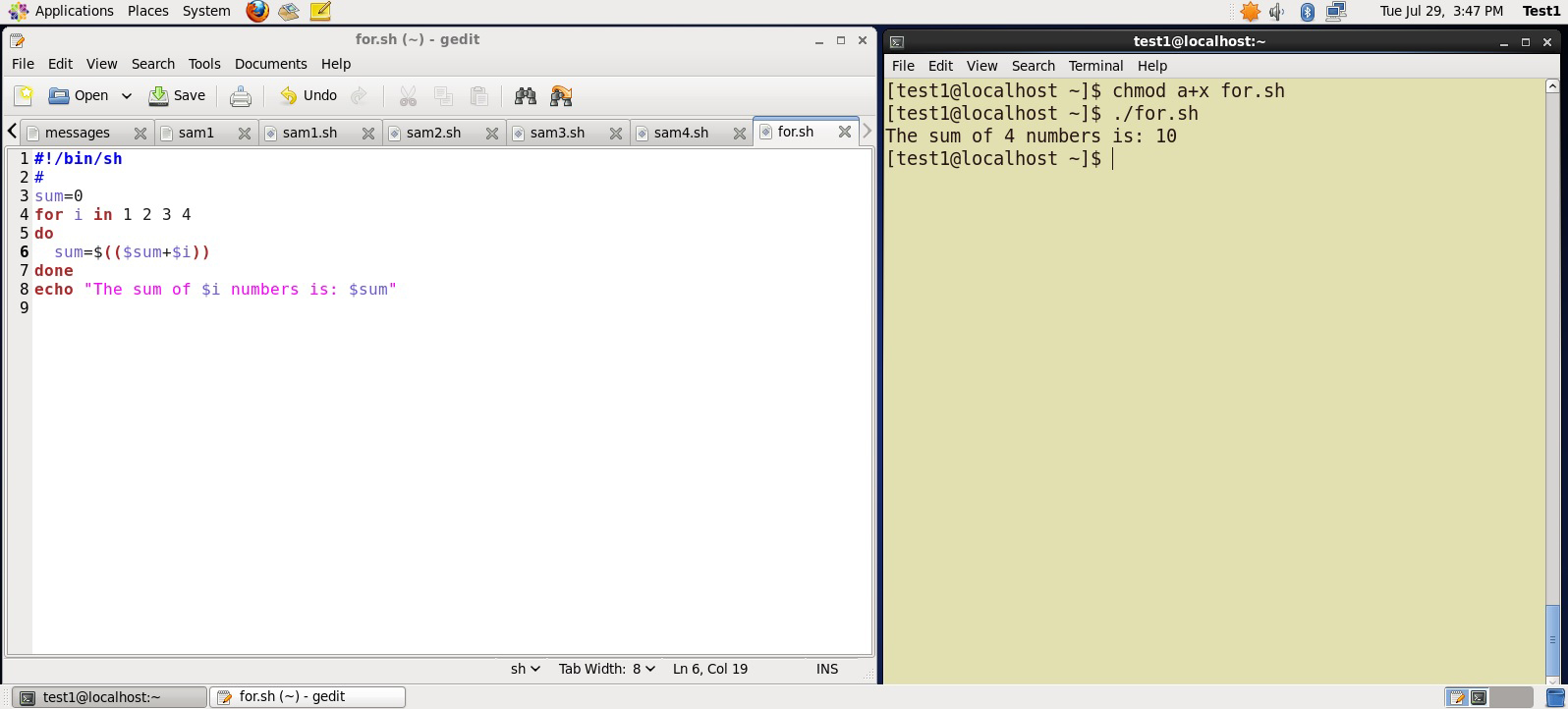
By using **looping constructs**, you can execute one or more lines of code repetitively. Usually you do this until a conditional test returns either true or false as is required.

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

* for
* while
* until

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

**The 'for' Loop**



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

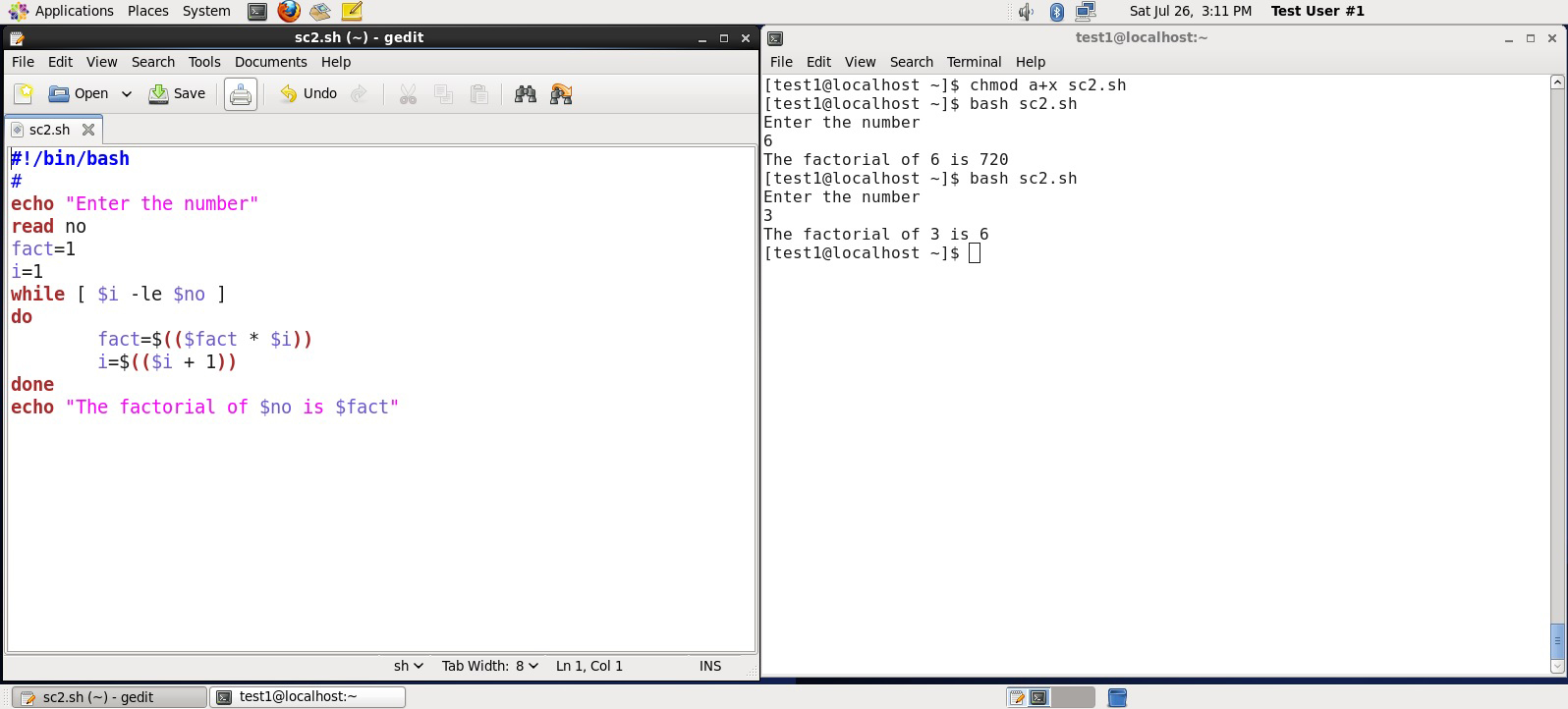
done

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**



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

----

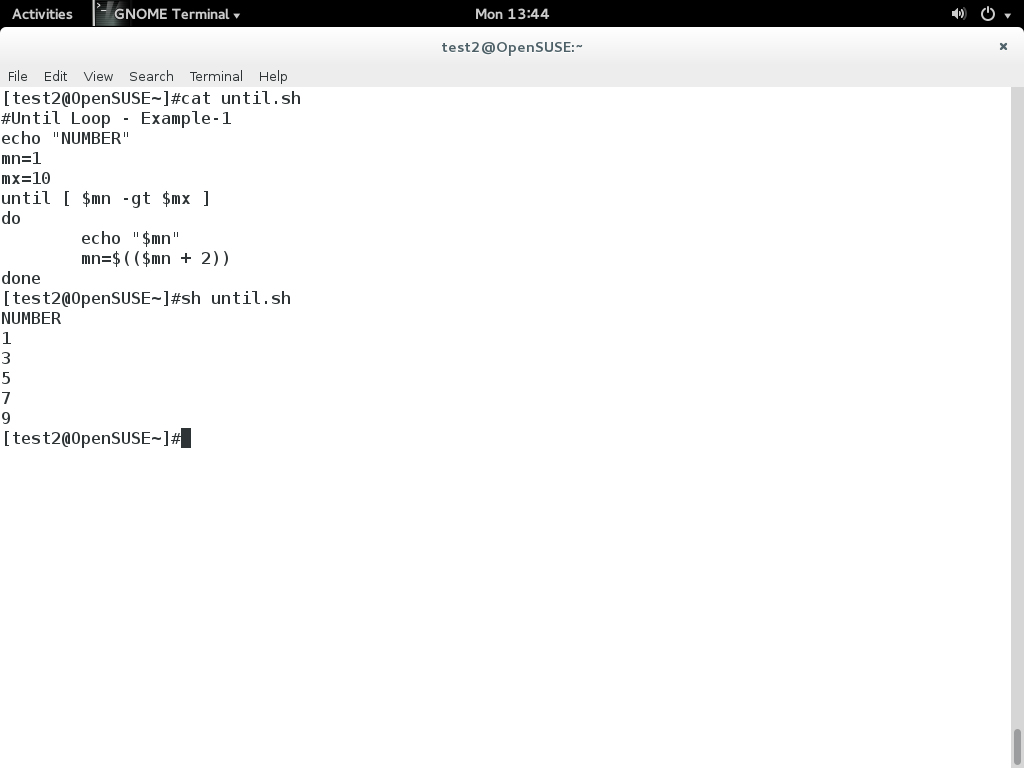
done

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**



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

until condition is false

do

Commands for execution

----

done

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

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

**Introduction to Script Debugging**



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

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

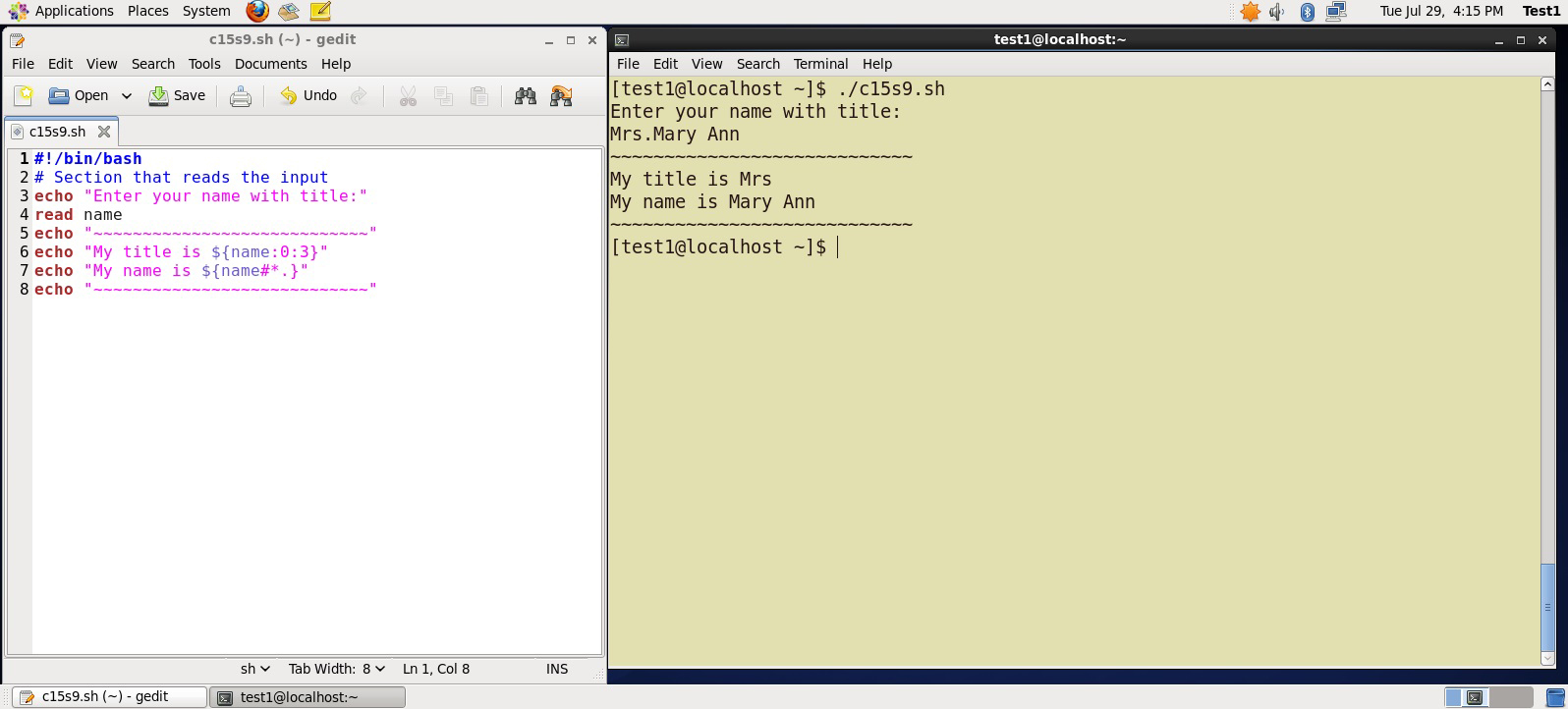
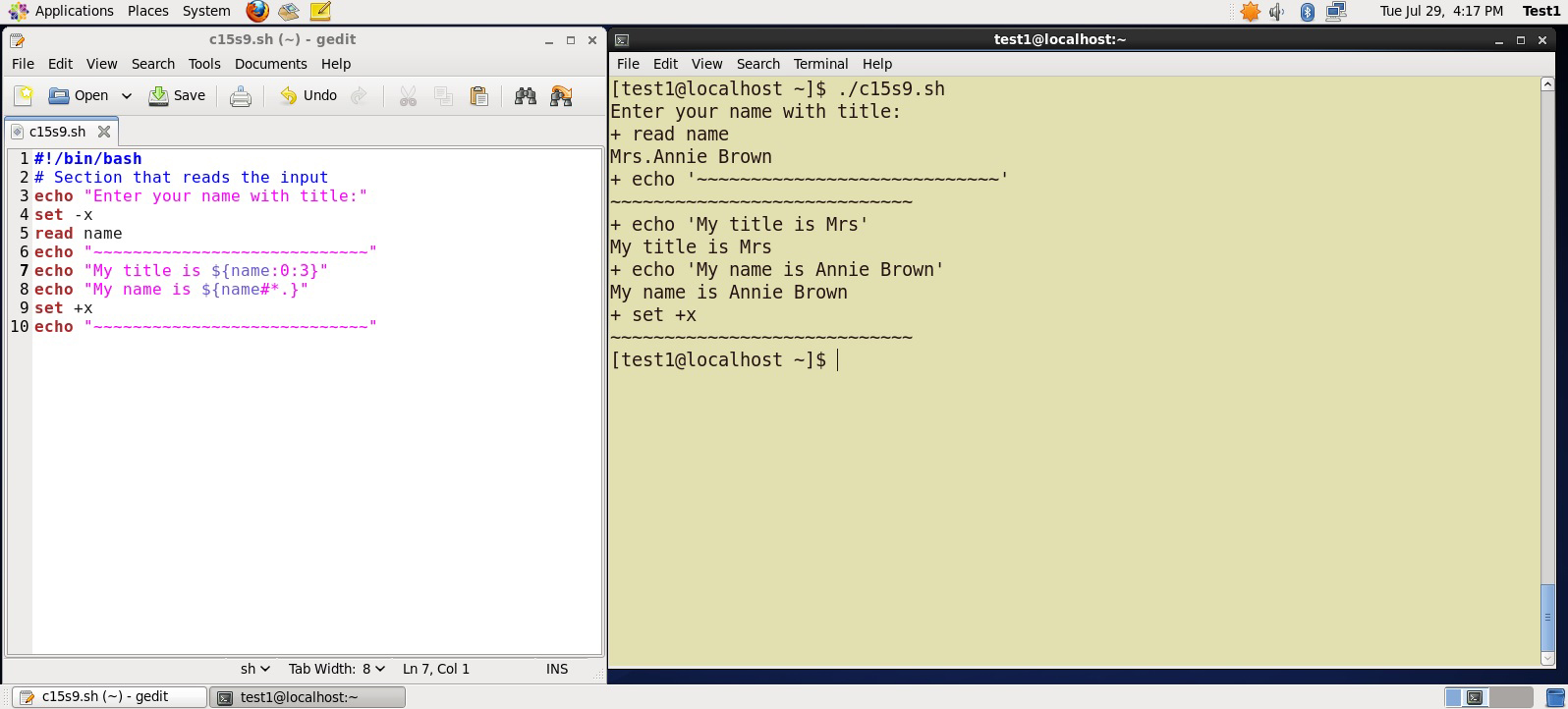
**More About Script Debugging**

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

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

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

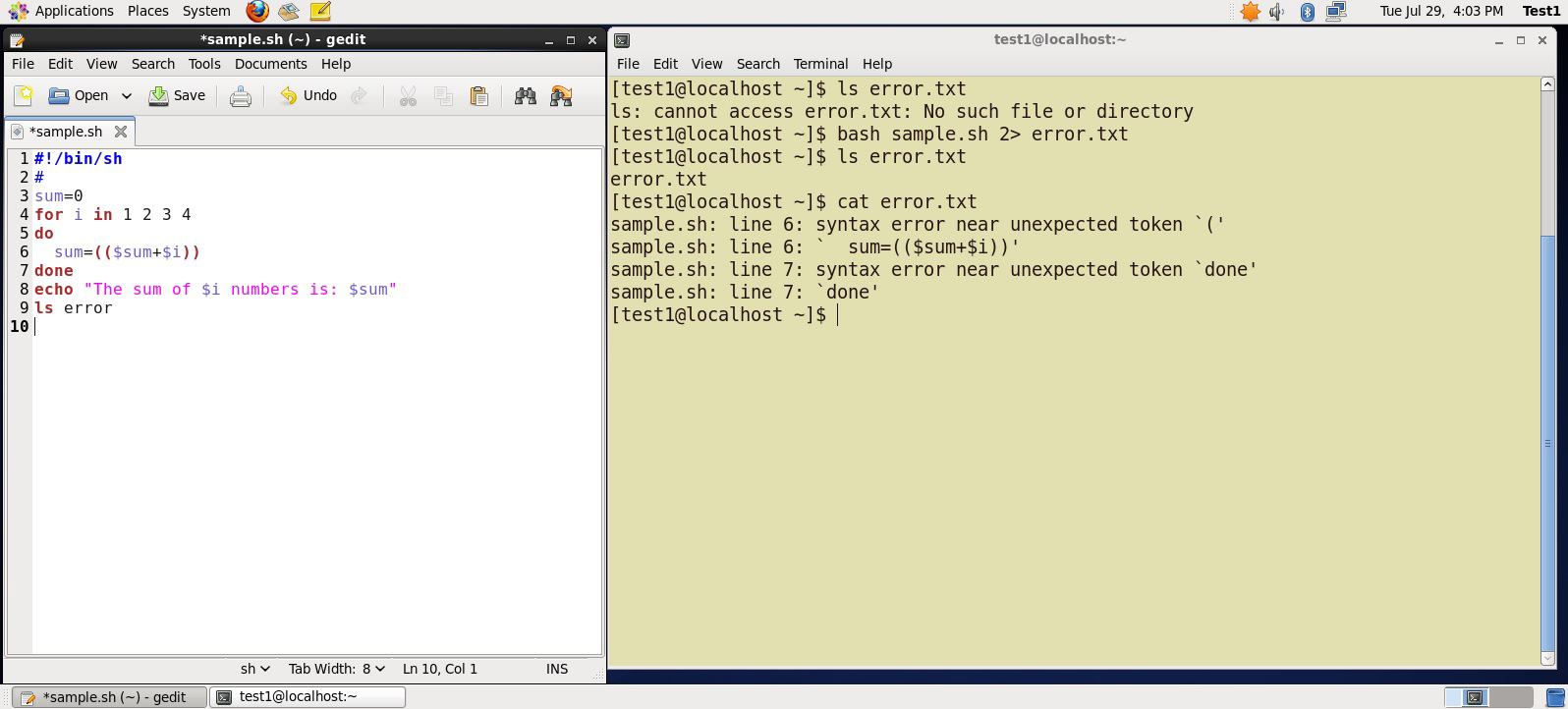
The screenshots shown here demonstrate a scriptfile called sc2 (left image) and the results from running it in debug mode (right image).



**Redirecting Errors to File and Screen**

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

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

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

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

Click the image to view an enlarged version.

**Creating Temporary Files and Directories**

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

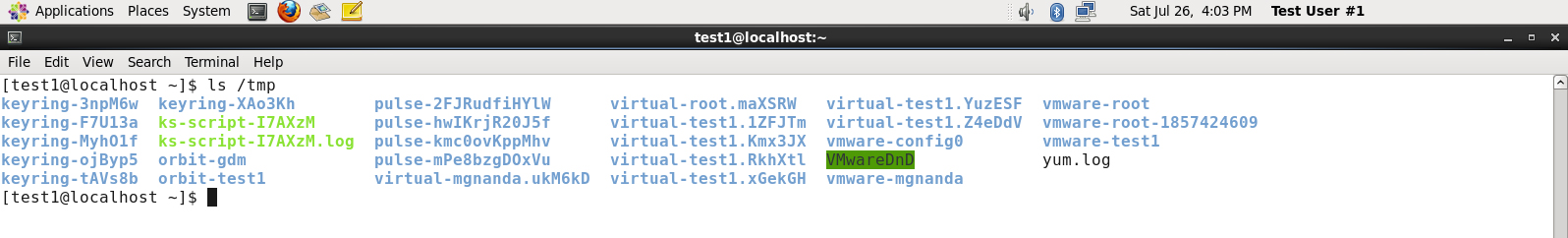
Temporary files (and directories) are meant to store data for a short time. Usually one arranges it so that these files disappear when the program using them terminates. While you can also use **touch** to create a temporary file, this may make it easy for hackers to gain access to your data.

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

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

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

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



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

$ ln -s /etc/passwd /tmp/tempfile

There could be a big problem if a script run by root has a line in like this:

echo $VAR > /tmp/tempfile

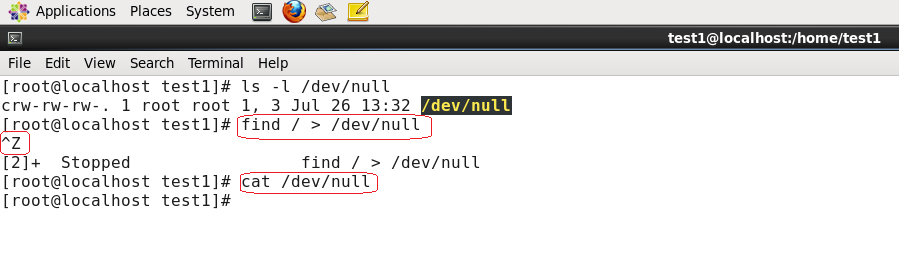
The password file will be overwritten by the temporary file contents.

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

TEMP=$(mktemp /tmp/tempfile.XXXXXXXX)

echo $VAR > $TEMP

**Discarding Output with /dev/null**

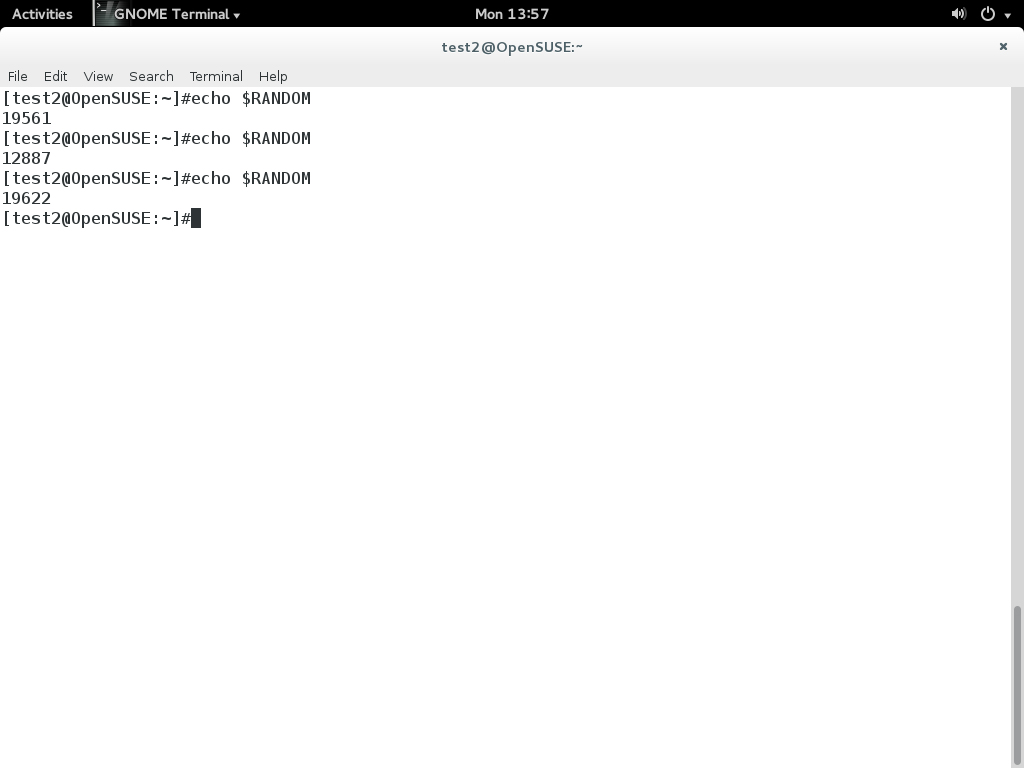
Certain commands like **find** will produce voluminous amounts of output which can overwhelm the console. To avoid this, we can redirect the large output to a special file (a device node) called **/dev/null**. This file is also called the **bit bucket** or **black hole**.

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

$ find / > /dev/null

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

**Random Numbers and Data**



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

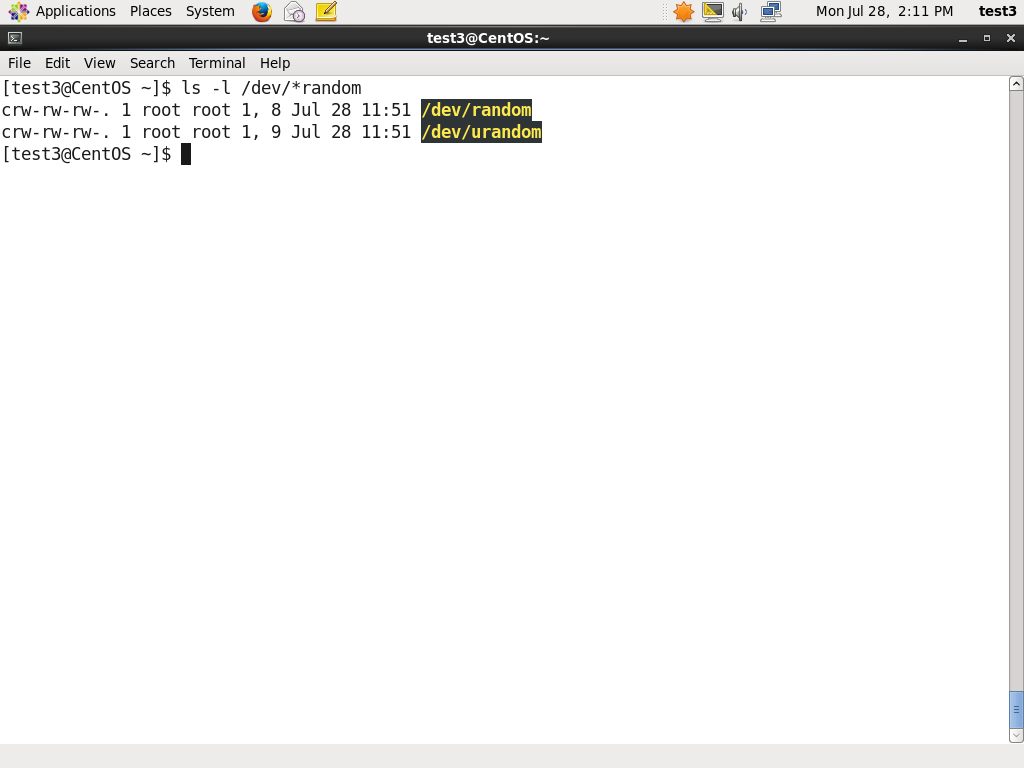
* Performing security-related tasks.
* Reinitializing storage devices.
* Erasing and/or obscuring existing data.
* Generating meaningless data to be used for tests.

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**



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.