

B. Sc. Computer III (Sem-VI)

Unit – I: Memory Management and Advanced VI

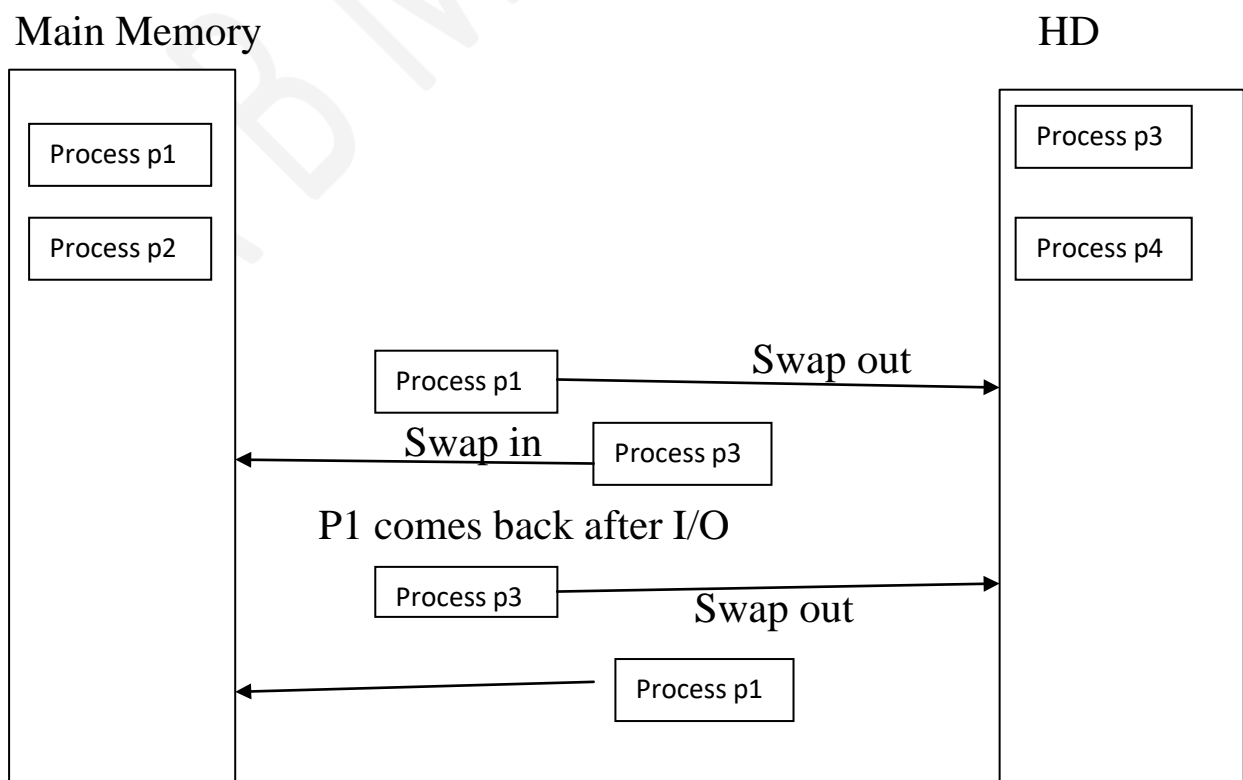
Memory Management

Memory Management is the functionality of an O.S. (kernel), which handles or manages primary memory & moves processes back and forth between main memory & disk(HD) during execution. Memory management keeps track of each & every memory location, where it is allocated to some process or it is free. It decides which process will get memory at what time. It tracks whenever some memory gets freed and correspondingly it updates the status.

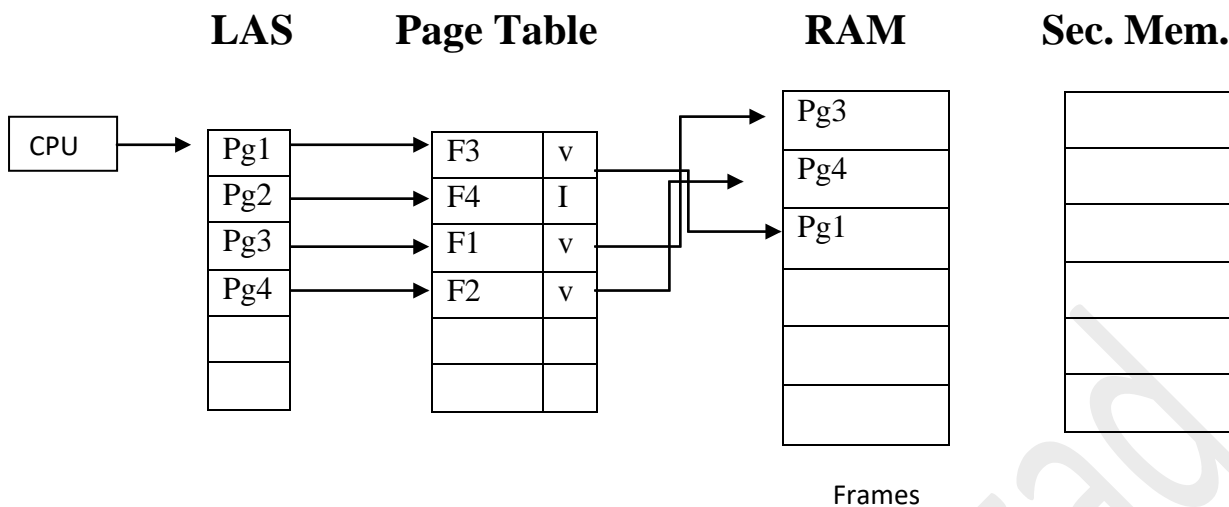
Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) & make that memory available to other processes which ready for execution. At some time, the system swaps back the process from the secondary storage to main memory whose finished I/P request.

Diagram



Demand Paging



In computer O. S. demand paging is a method of memory management. In system that uses demand paging, the O. S. copies a disk page into memory only if an attempt is made to access it and that page is not already in memory (i.e. if a page fault is occur).

Demand paging follows that pages should only be brought into memory if the executing process demands them. To achieve this process a page table implementation is used. The page table maps logical memory to physical memory. The page table uses a bitwise operator to mark if a page is valid or invalid. A valid page is one that currently resides in primary memory. (V=valid, I=Invalid)

When a process tries to access a page, the following steps are generally followed:

- *Attempt to access page.
- *If page is valid (in memory) then continue processing.
- *If page is invalid then a page fault occurs.
- *Check it the memory references are a valid reference to a location on secondary memory. If not, the process is terminated (illegal memory access). Otherwise, we have to page in the requested page.

*Schedule disk operation to read the desired page into main memory.

*Restart the instruction that was interrupted by the O. S. trap.

Advanced VI

Operators

Most of the useful functions in the command mode are derived from a combination of an operator & command; with operators you can handle any situation.

There are few operators.

1) d – delete 2) y – Yank (copy)

3) f – filter to act on text

Combination of operator & command perform any function.

Deleting & Moving text (d, p and P) [cut & paste]

VI can perform deletion when this operator is combined with a command of the command mode.

Consider that you have to delete text from the present cursor position to the end of line.

\$ command move to end of line.

You can combine the **d** operator with \$ to delete from the cursor position to the end of line.

d\$ delete rest of line.

Moving Text (p and P)

We deleted text using the **dd** & then placed the deleted text at a different location with the help of **p** and **P** commands.

p Puts text below the cursor.

P Puts text above the cursor.

Yanking Text (y)

The **y** operator yanks (or copies) text.

You copies a word (**yw**), line (**yy**) or (**yG**) entire lines. The **p** and **P** commands act in the same way as (below & above) for putting the copied text at its destination.

Filtering Text

VI has a feature of allowing a Linux filter to act on text, i.e. use the **!** operator & the sort command.

To filter (sort) text on the screen requires three steps:

a) Move to the beginning of the text to be acted upon & type **!**

b) Move to the other end of the text using any navigation like **G**

c) Enter the command to act on the text. The text on the screen should change immediately.

For example, to sort lines 1 to 4 of a file, move the cursor to the 1st character of line 1 & then press the following sequence.

```
:!4G sort
```

Another operators, the **!** when doubled, use the command following it to operate on the current line. Use **tr** to change the case of text of the current line to upper.

```
:!! tr „[a-z]“ „[A-Z]“
```

The Ex-Mode

Handling multiple files:

Inserting file & command output

In windows to insert contents of one file to another file; you have to open the file, use select all from the Edit menu, and copy the entire text with [Ctrl + C] switch to the original file and paste the contents with [Ctrl + V].

In VI, you don't have to visit other files. Just insert its contents at the present cursor location.

`:r note`

Insert file note1.

Command Output

You can also put the output of command in your file. Use `:r` like before, but instead of specifying the filename, enter the command name presented by !

`:r !date`

`:r !cal`

Insert output of date command.

The Ex-mode offers facilities related to handling multiple files

a) Switching from one file to another.

b) Using VI with multiple file names.

a) Switching from one file to another

Sometimes need to ignore all the unsaved changes you made and to switch to another file. You can edit multiple file without leaving the

editor. While editing one file, you can easily switch to another by using `:e` command.

`:e note 2`

You can return to the original file by using one of these sequences:

`[Ctrl + ^]` or `:e#` Toggle between current & previous.

b]Using VI with multiple filenames

When VI is used with multiple file, say **vi note [1-4]**, it loads the first file (note) into buffer. You can switch to the next file, note2 by using

`:n` next

Without saving

`:n!`

You can move back to the first file by making

`:rew`

Unit - 2:Advanced Filters

Advanced filters don't belong to the do one thing family of unix command. In fact, it can do several things. It operates at the field level and can easily access, transform and format individual field in a line. It also accepts extended regular expression for pattern matching, has C type programming constructs, like variables & several build in functions.

sed

It is derived from ed, the original UNIX editor. sed perform non-interactive operations on data stream. It uses very few options but has no. of features.

Syntax:

sed options 'address option' file(s)

The option provides various facilities & file is name of source file(s)

sed uses instructions to act on text. An instruction combines an address for selecting lines, with an action to be taken. The address and action are enclosed with in single quotes.

Address specifies either one line no. to select or set of two (3,7) to select group of Configuration Lines.

Line addressing

To consider line addressing first, the instructions, 3q it can be broken down to the address 3 and action „q“ (quit). When this instruction is enclosed within quotes and followed by one or more filenames.

```
$sed „3q“ emp
123 | ram | 30000
432 | sham | 15000
```

489 | yash | 20000

Generally use **p** (print) command to display lines. But its o/p, both the selected lines as well as all lines. So the selected lines appear twice. We must suppress this behavior with the **-n** option.

Selecting lines from anywhere

sed can also select a continuous group of lines from anywhere in the file. To select lines through 11, you have to use the following command:

```
$sed -n „9, 11p“ emp
```

Multiple instructions (-e, -f)

sed is not restricted to select as many sections from just about anywhere.

```
$sed -n -e „1, 2p“ -e „3, 5 p“ -e „7, 9p“ emp
```

When you have set of common instructions that you execute, they are better to store in a file. For instance, the above three instruction can be stored in a file with each instruction on a separate line.

```
$cat instr.txt
```

```
1,2p
```

```
3, 5p
```

```
7, 9p
```

You can now use the **-f** option to direct sed to take its instructions from the file using the command.

```
$sed -n -f instr.txt emp
```


Context Addressing

In context addressing, you specify one or two pattern, to locate lines. The pattern must be bounded by a / on either side, when you specify a single pattern, all line containing the pattern are selected.

```
$sed -n ,/director/p" emp
```

```
9876 | sharma | director | 7000
```

```
6521 | Rajput | director | 8000
```

You can also specify a comma to select group of lines.

```
$sed -n ,/patil/,/mane/p" emp
```

Internal Commands used by sed (i, a, d, p, r, w, q, s)

Writing selected lines to a file (w)

You can use the **w** (write) command to write the selected lines to a separate file. Save the lines in directories in dlist in this way.

```
$sed -n ,/director/w dlist emp
```

Inserting lines (i)

The **i** option insert text; assume C programmer want to add “include”, lines at the beginning of a program, prog (all ready existing file) in this way.

```
$sed ,1i/#include <stdio.h>/" prog
```

```
#include<stdio.h>
```

```
123 | ram | 20000
```

```
456| yash | 30000
```

Append (a)

The command used sed to append a data.

Example: Here it add “Hi” end of 1st line.

```
$sed „1a /Hi/“ prog
```

```
123 / ram / 20000
```

```
Hi
```

```
456 | yash | 30000
```

Change (c)

Here using C command change 1st line to “hello”

```
$sed „1c / Hello/“ prog
```

```
Hello
```

```
456 | yash | 30000
```

Deleting Lines (d)

Sed uses the d(deletes) command to delete selected lines.

```
$sed „,/director/d“ emp > dlist
```

Substitution (s)

Substitution is feature of sed. It replace a pattern in its input with something else.

Syntax:

```
[address]s/expr1/exp2/flags
```

Here, expression1 is replaced with expression2 in all lines specified by [address]

Example:

We used global substitution (**g**) to replace, all pipes (|) with colors. Use 1, \$s (entire file) or 1, 3\$ (1st three lines)

```
$sed „1,3s/|/:/g“ emp
```

It also applicable with string. lets replace the word dir to admin in the 1st five line of emp.

```
$sed „1,5s / dir / admin/“ emp
```

```
123 | ram | admin | 20000
```

Containing a different string (p)

It widens the scope of substitution. It is possible that you may like to replace a string in all lines containing a different string.

```
$sed -n „/marketing / s / dir / member / p“ emp
```

```
623 | rakesh | member | marketing | 20000          original line
```

```
623 | rakesh | dir | marketing | 20000             substituted line
```

gawk

gawk is a powerful language to manipulate and process text files. It is especially helpful when the lines in a text files are in a record format. i.e. A record containing multiple fields separated by a delimiter. Even when the input file is not in a record format, you can still use gawk to do some basic files and data processing. You can also write programming logic using gawk.

Syntax:

```
gawk options „selection_criteria {action}“ file(s)
```

The selection criteria filters i/p and selects lines for the action component to act upon. The action component is enclosed within curly braces. The selection criteria and action is surrounded by a set of single quotes.

The following command selects the director from emp.

```
$gawk „/director/ {print}“ emp
```

The selection criteria (/director/) selects lines that are processed action ({print})

gawk – field level operations

gawk perform field level operations using comparison operators and logical operators.

1) Comparison Operators

How do you print the field three for the director and chairman? Since the designation field is \$3, you have to use it in the selection criteria.

```
$gawk -F “|” „$3==“director” || $3==“Chairman”  
    {print “%-20s% - 12s%d\n”, $2,$3,$6}“ emp
```

This is the 1st time we matched a pattern with a field.

Here, gawk uses the || and && logical operator in the same sense as used by C.

For neglecting the above condition, you should use the !=

```
$gawk -F “|” „$3!=“c.m.” {printf “%20s %d\n” $2, $5}“ emp
```

~ and !~

The regular expression operators for matching a regular expression with a field, gawk offers the ~ and !~ operators to match & negate.

```
$gawk -F "|" ,,$2 ~/[Pp]atil/||/sale/{printf "%-20s%d\n" $2,$3}" emp
```

2)Number Comparison

gawk can also handle number both integer and floating type and make relation tests on them.

e.g. you can now print pay slips for those people where basic pay exceeds 7500.

```
$gawk -F "|" ,,$6>7500 {printf "%-20s%-12s%d\n", $2,$3,$6}" emp
```

Formatted output: printf

Because of this data is unformatted but with the C like printf statement, you can use gawk as stream formatter. gawk accepts most of the formats used by the printf function in C. Here %s is for string data, %d for integer data & %f for float data.

e.g. produce list of "rakesh" with formatting.

```
$cat emps
```

```
123 | rakesh | 12000 .50
```

```
345 | ramesh | 15000.75
```

```
$gawk -F "|" ,,/rakesh/{printf "%-10d %20s      %f\n" $1,$2,$3}  
emps
```

```
123                rakesh                1234.440000
```

This id & name have been printed in space 10 and 20 character wide. „-, symbol left justified the o/p, name field is 20 character wide but it is right justified. Also user can use \n for add new line for tab.

Use of Variables & Expressions

We can use variables & expressions with gawk. Expressions contains numbers variables & operators i.e. (x+5)*12. gawk doesn't have char, int, long data types. Every expression can be interpreted either as a string or a number.

gawk allows the use of user defined variables but without declaring them. Variables are case sensitive; a is different from A.

```
$gawk „x=5 {printf “%d \n”, x}“
```

String in awk are always double quoted and can contain any characters.

```
$gawk „x= \t\tchadgad” {printf “%-15s\n”, x}“
```

gawk also concatenate the strings, just you have to place it side by side.

```
$gawk „x=“Tal.”; y=“chadgad”; {print x y}“
```

Every expression can be interpreted either as a string or number, and gawk makes the necessary conversion according to content.

```
x = “5”; y = 6; z = “A”
```

```
print x, y      y converted to string; prints 56
```

```
print x + y     x converted into number; print 11
```

```
print y+z       z converted to numeric 0; print 6
```

```
$gawk „x=“5.”; y = “6”; {printf x + y}“
```

The BEGIN and END section

If you have to print something before processing the 1st line, like: a heading, then the BEGIN section can be used quite gainfully, similarly, the END section is useful in printing something like: totals after processing is over.

```
BEGIN {action}
```

```
END {action}
```

These two sections, are delimited by the body of the gawk program, gawk also uses the # for providing comments.

```
$cat emp.gawk
```

```
BEGIN
```

```
{  
  printf "\t\t Employee Abstract \n\n"  
}
```

```
$5>40000
```

```
{  
  Count ++;  
  Tot = Tot + $5  
  printf "%3d%-20s\n", count, $2  
}
```

```
END
```

```
{  
  printf "\n\tThe average basic salary is %6d \n", Tot/Count  
}
```

\$cat aw

2222 | Sunil |g.m. | sales | 45400

2323 | ram |d.m. | accou | 46600

2234 | Sham |c.m. | man | 34400

\$gawk -F "|" -f empawk aw

Employee Abstract

1 sunil

2 ram

The average basic salary is 46000.

Built – In variables

gawk has several built in variables.

They are all assigned automatically. It is also possible user to reassign.

1)NR

It show the record no. of the current line.

\$ gawk -F "|" ,NR==3, NR==5 {print NR, \$2, \$3, \$3,\$6}" emp

3 ram 20000

4 yash 30000

5 Rakesh 25000

The statrement NR ==3 is a condition that is being tested.

2) FS

gawk uses continuous string of spaces as the default field separator. FS redefined this field separator. It mostly occur in the BEGIN

section. So that the body to the program known its value before it start processing. This is an alternative to the -F optional which does the same thing.

```
$gawk    „BEGIN {FS = “|”}
```

```
>NF != 6
```

```
>{print “Record No.”, NR, “has”, NF, “fields”}“ emp
```

```
Record No. 6 has 4 field
```

```
Record No. 8 has 5 field
```

3) The FILENAME variable

FILENAME stores the name of the current file being processed. By default, gawk doesn't print the filename, but you can instruct it to do so.

```
$gawk -F “|” „$1<300 {print FILENAME, NR, $2,$3}“ emp
```

```
emp    1    ram    dir
```

4) The NF variables

The NF is useful for cleaning up a database of lines does not contain the right no. of fields.

Example: To locate those line not having six fields, due to faculty data entry.

Built in variable used by gawk:

Variable

Function

NR

cumulative no. of lines read.

FS	I/P field separator
OFS	O/P field separator
NF	No. of field in current line.
FILENAME	Current Input file.
ARGC	No. of argument in command line.
ARGV	List of argument.

Arrays

An array is also a variable except that this can store a set of values or elements; each element is accessed by a subscript called the index.

Rules of arrays:

- 1) They are not formally defined.
- 2) Array elements are initialized to zero or an empty string unless initialized explicitly.
- 3) Array expanded automatically.
- 4) The index can be virtually anything; it can be even be a string.

In the program empawk; we use arrays to store the totals of the basic pay, da, hra and gross pay of the sales and marketing people.

Assume that the da is 25%, hra 50% of basic pay, use the tot[] array to store the totals of each elements of pay and also gross pay.

```
$cat empawk1
```

```
BEGIN
```

```
{
    FS = "|"
```

```
printf "%46s \n", "Basic, DA, HRA, Gross"
```

```
}/sales |char/{
```

```
da = 0.25 * $5; hra = 0.50 * $5;
```

```
gp = $5 + hra + da
```

```
tot [1] += $5;
```

```
or tot [1] = tot[1] + $5;
```

```
tot[2] += da;
```

```
to[3] += hra;
```

```
tot[4] += gp;
```

```
count ++;
```

```
}
```

```
END
```

```
{
```

```
printf "Average %5f %5f %5f %5f \n",
```

```
tot[1]/count, tot[2] / count
```

```
tot[3]/count, tot[4] / count
```

```
}
```

```
$awk -F "|" -f empawk1 aw
```

	Basic	DA	HRA	Gross
--	-------	----	-----	-------

Average	399.00	99.75	199.50	698.25
---------	--------	-------	--------	--------

Functions

gawk has several built in functions performing both arithmetic & string operation. The argument are passed to function in C style separated by commas & enclosed by pair of parentheses.

1)length

Length determines the length of its argument, & if no argument is present, the entire line is assumed to be the argument.

without argument: To locate lines whose length exceeds 1024 char.

```
$gawk -F "|" length>1024" emp
```

with arguments: To selects those people who have short name.

```
$gawk -F "|" length($2) < 11" emp
```

2)substr

The substr (stg,m,n) function extracts a substring from a string stg. m represents the starting point of extraction & n indicates the no. of character to be extracted.

e.g. To select those born between 1946 & 1951.

```
$gawk -F "|" „substr($4,7,2) >45 && substr($4,7,2) < 52" emp
```

```
2565 | ram |director |11/05/47 |30000
```

3)split

The split (stg,ar,ch) breaks up a string stg on the ch & stores the fields in an array ar[]. Here's how you can convert the date field to the format YYYYDDMM

```
$gawk -F "|" {split($4,ar,"/"); print "19" ar[3]ar[2]ar[1]}" emp
```

4)system

You want to print the system date at the beginning of the report.

e.g. `BEGIN {system("date")}`

Types of Meta Character

The special set of character that the shell uses to match filenames. Previously you used commands with more than one filename as arguments (e.g. `$cat stud01 stud02`). Where, you may need to enter multiple filenames in a command line. To get clear idea try to listing all filenames beginning with stud. The most obvious solution is

```
$ls stud01 stud02 stud03 stud04 stud05
```

If the filename are similar, we use the facility offered by shell of representing them by single pattern. For example `stud*` represents all filenames beginning with stud. This pattern contains character (like stud) and metacharacter (like *). Here, shell will expand it before the command is executed. The metacharacter that are used to make pattern for matching filenames belonging to a category called wild-cards.

Wild-Card

Methods

*	Any number of characters including null
?	A single character
[ijk]	A single character – either an i, j ,k
[x-z]	A single character within ASCII range x To z
[!ijk]	A single character that is not an i, j ,k
[!x-z]	A single character not within ASCII range x To z

UNIT 3: ADVANCED SHELL PROGRAMMING

Shell and Subshell

When the shell executes a shell script, it first spawns a subshell, which again executes commands in the script. When script execution is complete, the child returns control to the parent shell.

Ex. Subshell used to read the statements in if1.sh

```
$sh if1.sh
```

Thus a shell script run with sh need not have execute permission. This is applicable for executing only shell scripts.

Above ex. We specify script name from the shell prompt & uses a sub shell of the same type.

If the script contains the interpreter lines in this form.

```
#!/user/bin/ksh
```

Then, even though the login shell may be Bash, it will use the korn shell to execute the script.

Command Line Arguments (Shell Variable)

The shell supports argument that are useful both in the command line and shell script. They having there predefined task and its evaluation requires the \$ as prefix to the variable name.

\$PATH = shows set of directories where executable programs are located.

\$SHELL = It shows current working shell.

\$# = The no. of argument supplied to the script.

\$* = It shows argument that specified.

\$\$ = The process id of the current shell

\$! = The process id of the last background job.

\$? = Exit status of last command.

\$O = The file name of the current process.

```
$cat script3
```

```
echo "program:" $0
```

```
echo "The no. of argument specified is", $#
```

```
echo "The arguments are" $*
```

```
grep "$1" $2
```

```
echo "\n Job Over"
```

```
$sh script3 director emp
```

```
program : script3
```

```
The no. of argument specified is 2.
```

```
The arguments are director emp
```

```
1000 | Ram | director | sales
```

```
3010 | Yash | director | HR
```

Exporting Shell Variables

```
$cat expt.sh
```

```
echo "The value of x is" $x
```

```
x=20
```

```
echo "The value of x is " $x
```

```
$sh expt.sh
```

The value of x is

The value of x is 20

```
$x = 10; sh expt.sh
```

The value of x is

The value of x is 20

```
$echo $x
```

10

```
$x = 10; export x
```

```
$echo $x
```

10

```
$sh expt.sh
```

The value of x is 10

The value of x is 20

By default the values stored in shell variables are local to the shell & are not passed on to a child shell. But the shell can also export these variables to all child processes, once defined, they are available globally.

We assigned 10 to x at the prompt & then execute the script : `$x = 10; sh empl.sh`

The value of x is

The value of x is 20.

Because x is a local variable in the login shell, its value can't be accessed by echo in the script, which it run in sub shell. To make an x

available globally, you need to use export statement before the script is executed.

i.e. `$x = 0; export x`

when x is executed, its assignment value (10) is available in the script. But when you export a variable, it has important thing, a reassignment (`x=20`) made in the script (a sub shell) is not seen in the parent shell which executed the script.

Shell Functions

A shell function is like any other function; it executes a group of statements as bunch. It also returns a value. This is useful to perform repetitive task.

Syntax

```
Function_name()  
{  
    Statements  
    Return value;  
}
```

The function definition is followed by (), & the body is enclosed within curly braces. On calling, the function executes all statements in the body. The return statement, when present, returns a value.

Ex. To view no. of files in a directory,

```
$cat nofiles()  
  
{  
  
ls    -l    $x  
  
}
```

Like shell scripts, shell functions also use command like arguments (like \$1,\$2), We use () when we define function, but it is not use when invoking the function with or without arguments.

\$nofiles

set

set command assigns its arguments to the positional parameters \$1, \$2 and so on. This feature is especially useful for picking up individual fields from the o/p of a program.

a) set to convert its arrangement to positional parameter:

```
$set      9876      2345      6213
```

This assigns the value 9876 to the positional parameter \$1, 2345 to \$2 & 6213 to \$3.

```
$echo      “\ $1 is $1, \ $2   is $2, \ $3 is $3”
```

\$1 is 9876, \$2 is 2345, \$3 is 6213.

b) Extracts individual fields from the date o/p

```
$set      „date“
```

```
$date
```

```
Wed      Jan   24   09:40:35   IST   2016
```

```
$echo      “The date today is $2, $3, $6”
```

The date today is Jan 24, 2013

Data Validation before storing outside

In valcode.sh script, it accept I/P from the user. Script looks up a code list for data validation.

This script, valcode.sh use the set features to accept & validate a department code. It looks up a code list maintained here as a document in the script file itself & display the department name on the terminal.

The pattern selected by grep is split by set on the | into three positioned parameters. This is done by changing the IFS setting that normally consists of white space.

```
$cat valcode.sh

IFS = "|"

while echo "Enter department code:"
do
read dcode
set -- `grep "$dcode" <<limit
01 | sales      | 6213
02 | HR         | 5423
03 | production | 6521
Limit`
case $# in
3) echo "Department name: $2 \nEmp -id of the head of dept: $3\n";;
*) echo "Invalid code"; continue
esac
```

done

\$ssh valcode.sh

Enter department code :99

Invalid code

Enter department code : 02

Department Name : HR

Emp id of Head of Dept: 5423

Enter department code: (ctrl + c)

Write data entry script to create data files.

In a script, dentry.sh accepts a designation code its description from the terminal, performs validation checks, & then adds an entry to a file (design.sh). It check the code entered with the already exist in the file.

The script repeatedly prompts the user until the right response is obtained.

The script prompts for two files, the designation code and description and use two while loops, one enclosed by the other.

The code has to be reentered if it exists in the file or if it doesn't have a two digit structure. Similarly, the description has to be reentered if it contains a non-alphabetic character other than a space (*[!a-z,A-Z]*). The continue statements let you start a fresh cycle. The break statement in the inner loop.

\$cat design.sh

01 | accounts

02 | admin

03 | marketing

\$sh dentry1.sh

Desigcode : 01

Code code : 01

Code exist

Desig code : 04

Description : Security officer

Wish to continue (y/n):y

Desig code : 5

Invalid code

Desig code : 05

Description : vice president1

Can contain only alphabets & spaces

Description: vice president

Wish to continue? (y / n) : n

Normal exit.

\$cat dantryl.sh

File = design.lst

while echo "Desig code."; do

read design

```
case "$desig" in
[0-9] [0-9] ) if grep "^$desig" $file
then
echo "code exist"
continue
fi;;
*) echo "Invalid code"
continue
esac
while echo "Description", ; do
    read desc
    case "$desc" in
        *[\a-z A-Z]* ) echo "Can contain only alphabets & spaces"
Continue;;
        " ") echo "Description not entered"
Continue;;
    esac
done>>$file
echo "Wish to continue?" (y/n)
read answer
case "$answer" in
[yY]*) continue;;
```

```
*) break;;
```

```
esac
```

```
done
```

```
echo "Normal exit."
```

Unit - 2: Advanced Filters

Advanced filters don't belong to the do one thing family of unix command. In fact, it can do several things. It operates at the field level and can easily access, transform and format individual field in a line. It also accepts extended regular expression for pattern matching, has C type programming constructs, like variables & several build in functions.

Sed

It is derived from ed, the original UNIX editor. sed perform noninteractive operations on data stream. It uses very few options but has no. of features.

Syntax:

sed options 'address option' file(s)

The option provides various facilities & file is name of source file(s)

sed uses instructions to act on text. An instruction combines an address for selecting lines, with an action to be taken. The address and action are enclosed with in single quotes.

Address specifies either one line no. to select or set of two (3,7) to select group of Configuration Lines.

Line addressing

To consider line addressing first, the instructions, 3q it can be broken down to the address 3 and action „q“ (quit). When this instruction is enclosed within quotes and followed by one or more filenames.

```
$sed „3q“ emp
```

```
123 | ram | 30000
```

```
432 | sham | 15000
```

```
489 | yash | 20000
```

Generally use **p** (print) command to display lines. But its o/p, both the selected lines as well as all lines. So the selected lines appear twice. We must suppress this behavior with the **-n** option.

Selecting lines from anywhere

sed can also select a continuous group of lines from anywhere in the file. To select lines through 11, you have to use the following command:

```
$sed -n „9, 11p“ emp
```

Multiple instructions (-e, -f)

sed is not restricted to select as many sections from just about anywhere.

```
$sed -n -e „1, 2p“ -e „3, 5 p“ -e „7, 9p“ emp
```

When you have set of common instructions that you execute, they are better to store in a file. For instance, the above three instruction can be stored in a

file with each instruction on a separate line.

```
$cat instr.txt
```

```
1,2p
```

```
3, 5p
```

```
7, 9p
```

You can now use the **-f** option to direct sed to take its instructions from the file using the command.

```
$sed -n -f instr.txt emp
```

Context Addressing

In context addressing, you specify one or two pattern, to locate lines. The pattern must be bounded by a / on either side, when you specify a single pattern, all line containing the pattern are selected.

```
$sed -n ,/director/p" emp
```

```
9876 | sharma | director | 7000
```

```
6521 | Rajput | director | 8000
```

You can also specify a comma to select group of lines.

```
$sed -n ,/patil/,/mane/p" emp
```

Internal Commands used by sed (i, a, d, p, r, w, q, s)

Writing selected lines to a file (w)

You can use the w (write) command to write the selected lines to a separate file. Save the lines in directories in dlist in this way.

```
$sed -n ,/director/w dlist emp
```

Inserting lines (i)

The i option insert text; assume C programmer want to add “include”, lines at the beginning of a program, prog (all ready existing file) in this way.

```
$sed ,,1i/#include<stdio.h> /" prog
```

```
#include <stdio.h>
```

```
123 | ram | 20000
```

```
456| yash | 30000
```

Append (a)

The command used sed to append a data.

Example: Here it add “Hi” end of 1st line.

```
$sed ,,1a /Hi/" prog
```

```
123 / ram / 20000
```

```
Hi
```

```
456 | yash | 30000
```

Change (c)

Here using C command change 1st line to “hello”

```
$sed ,,1c / Hello/" prog
```

Hello

456 | yash | 30000

Deleting Lines (d)

Sed uses the d(deletes) command to delete selected lines.

```
$sed „/director/d“ emp > dlist
```

Substitution (s)

Substitution is feature of sed. It replace a pattern in its input with something else.

Syntax:

```
[address]s/expr1/exp2/flags
```

Here, expression1 is replaced with expression2 in all lines specified by [address]

Example:

We used global substitution (g) to replace, all pipes (|) with colors. Use 1, \$s (entire file) or 1, 3\$ (1st three lines)

```
$sed „1,3s/|/:/g“ emp
```

It also applicable with string. lets replace the word dir to admin in the 1 st five line of emp.

```
$sed „1,5s / dir / admin/“ emp
```

```
123 | ram | admin | 20000
```

Containing a different string (p)

It widens the scope of substitution. It is possible that you may like to replace a string in all lines containing a different string.

```
$sed -n „/marketing / s / dir / member / p“ emp
```

```
623 | rakesh | member | marketing | 20000
```

original line

```
623 | rakesh | dir | marketing | 20000
```

substituted line

Grep and grep options

- The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (grep stands for globally search for regular expression and print out).

Syntax:

```
grep [options] pattern [files]
```

Options Description

- c** : This prints only a count of the lines that match a pattern
- h** : Display the matched lines, but do not display the filenames.
- i** : Ignores, case for matching
- l** : Displays list of a filenames only.
- n** : Display the matched lines and their line numbers.
- v** : This prints out all the lines that do not matches the pattern
- e exp** : Specifies expression with this option. Can use multiple times.
- f file** : Takes patterns from file, one per line.
- E** : Treats pattern as an extended regular expression (ERE)
- w** : Match whole word
- o** : Print only the matched parts of a matching line, with each such part on a separate output line.

- A n** : Prints searched line and nlines after the result.
- B n** : Prints searched line and n line before the result.
- C n** : Prints searched line and n lines after before the result.

gawk command in Linux with Examples

- Last Updated : 06 May, 2019

gawk command in Linux is used for pattern scanning and processing language. The [awk command](#) requires no compiling and allows the user to use variables, numeric functions, string functions, and logical operators. It is a utility that enables programmers to write tiny and effective programs in the form of statements that define text patterns that are to be searched for, in a text document and the action that is to be taken when a match is found within a line.

gawk command can be used to :

- Scans a file line by line.
- Splits each input line into fields.
- Compares input line/fields to pattern.
- Performs action(s) on matched lines.
- Transform data files.
- Produce formatted reports.
- Format output lines.
- Arithmetic and string operations.
- Conditionals and loops.

Syntax:

```
gawk [POSIX / GNU style options] -f progfile [--] file ...  
gawk [POSIX / GNU style options] [--] 'program' file ...
```

Some Important Options:

- **-f progfile, -file=progfile:** Read the AWK program source from the file program-file, instead of from the first command line argument. Multiple -f (or -file) options may be used.
- **-F fs, -field-separator=fs:** It uses FS for the input field separator (the value of the FS predefined variable).
- **-v var=val, -assign=var=val:** Assign the value *val* to the variable *var*, before execution of the program begins.

Examples:

- **-F:** It uses FS for the input field separator (the value of the FS predefined variable).

```
gawk -F: '{print $1}' /etc/passwd
```

```
linux@ubuntu:~/files$ gawk -F: '{ print $1 }' /etc/passwd
root
daemon
bin
sys
sync
games
man
lp
mail
news
uucp
proxy
www-data
backup
list
irc
gnats
nobody
systemd-network
systemd-resolve
syslog
messagebus
_apt
uidd
avahi-autoipd
usbmux
dnsmasq
rtkit
cups-pk-helper
speech-dispatcher
whoopsie
kernoops
saned
pulse
avahi
colord
lightdm
```

- **-f**: Read the AWK program source from the file program-file, instead of from the first command line argument. Multiple -f (or --file) options may be used.
gawk -F: -f mobile.txt /etc/passwd

```

linux@ubuntu:~/files$ gawk -F: -f mobile.txt /etc/passwd
root:x:0:0:root:/root:/bin/bash
root:x:0:0:root:/root:/bin/bash
root:x:0:0:root:/root:/bin/bash
root:x:0:0:root:/root:/bin/bash
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
sync:x:4:65534:sync:/bin:/bin/sync
sync:x:4:65534:sync:/bin:/bin/sync
sync:x:4:65534:sync:/bin:/bin/sync
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin

```

Some Built In Variables:

- **NR:** It keeps a current count of the number of input line.
- **NF:** It keeps a count of the number of fields within the current input record.
- **FS:** It contains the field separator character which is used to divide fields on the input line.
- **RS:** It stores the current record separator character.
- **OFS:** It stores the output field separator, which separates the fields when Awk prints them.
- **ORS:** It stores the output record separator, which separates the output lines when Awk prints them.

Examples:

- **NR:**
 gawk '{print NR "-" \$1 }' mobile.txt

```
linux@ubuntu:~/files$ gawk '{print NR "-" $1}' mobile.txt
1-Deepak
2-Sunil
3-Aman
4-Kuldeep
5-Sundip
linux@ubuntu:~/files$
```

- RS:

gawk 'BEGIN{FS=":"; RS="-"} {print \$1, \$6, \$7}' /etc/passwd

```
linux@ubuntu:~/files$ gawk 'BEGIN{FS=":"; RS="-"} {print $1,$6,$7}' /etc/passwd
root /root /bin/bash
daemon
data
data 34 backup
Reporting System (admin) 65534 nobody
network /run/systemd/netif /usr/sbin/nologin
systemd
resolve /run/systemd/resolve /usr/sbin/nologin
syslog
autoipd /var/lib/avahi
autoipd usbmux daemon,,, /var/lib/usbmux
pk
helper
pk
helper service,,,
pk
helper
dispatcher /var/run/speech
dispatcher /nonexistent
daemon colord colour management daemon,,, /var/lib/colord
initial
setup /run/gnome
initial
setup/ Gnome Display Manager /var/lib/gdm3
```

- OFS:

gawk 'BEGIN{FS=":"; OFS="-"} {print \$1, \$6, \$7}' /etc/passwd

```
linux@ubuntu:~/files$ gawk 'BEGIN{FS=":"; OFS="-"} {print $1,$6,$7}' /etc/passwd
root-/root-/bin/bash
daemon-/usr/sbin-/usr/sbin/nologin
bin-/bin-/usr/sbin/nologin
sys-/dev-/usr/sbin/nologin
sync-/bin-/bin/sync
games-/usr/games-/usr/sbin/nologin
man-/var/cache/man-/usr/sbin/nologin
lp-/var/spool/lpd-/usr/sbin/nologin
mail-/var/mail-/usr/sbin/nologin
news-/var/spool/news-/usr/sbin/nologin
uucp-/var/spool/uucp-/usr/sbin/nologin
proxy-/bin-/usr/sbin/nologin
www-data-/var/www-/usr/sbin/nologin
backup-/var/backups-/usr/sbin/nologin
list-/var/list-/usr/sbin/nologin
irc-/var/run/ircd-/usr/sbin/nologin
gnats-/var/lib/gnats-/usr/sbin/nologin
nobody-/nonexistent-/usr/sbin/nologin
systemd-network-/run/systemd/netif-/usr/sbin/nologin
systemd-resolve-/run/systemd/resolve-/usr/sbin/nologin
syslog-/home/syslog-/usr/sbin/nologin
messagebus-/nonexistent-/usr/sbin/nologin
_apt-/nonexistent-/usr/sbin/nologin
uidd-/run/uidd-/usr/sbin/nologin
avahi-autoipd-/var/lib/avahi-autoipd-/usr/sbin/nologin
usbmux-/var/lib/usbmux-/usr/sbin/nologin
dnsmasq-/var/lib/misc-/usr/sbin/nologin
rtkit-/proc-/usr/sbin/nologin
cups-pk-helper-/home/cups-pk-helper-/usr/sbin/nologin
speech-dispatcher-/var/run/speech-dispatcher-/bin/false
whoopsie-/nonexistent-/bin/false
kernoops-/-/usr/sbin/nologin
saned-/var/lib/saned-/usr/sbin/nologin
pulse-/var/run/pulse-/usr/sbin/nologin
avahi-/var/run/avahi-daemon-/usr/sbin/nologin
colord-/var/lib/colord-/usr/sbin/nologin
hplip-/var/run/hplip-/bin/false
```

Sample More Commands with Examples:

- Consider the following sample text file as the input file for all cases below.

To create a text file:

```
cat > mobile.txt
```

```
linux@ubuntu:~/files$ cat > mobile.txt
Deepak 9984537565
Sunil 8875546255
Aman 9865741259
Kuldeep 8875463125
Sundip 9822456788
```

- **Default behavior of gawk:** By default gawk prints every line of data from the specified file.

```
gawk '{print}' mobile.txt
```



```
linux@ubuntu:~/files$ gawk '{print}' mobile.txt
Deepak 9984537565
Sunil 8875546255
Aman 9865741259
Kuldeep 8875463125
Sundip 9822456788
linux@ubuntu:~/files$
```

- **To print the lines matching with the given pattern:**

```
gawk '/Sunil/ {print}' mobile.txt
```

```
linux@ubuntu:~/files$ gawk '/Sunil/ {print}' mobile.txt
Sunil 8875546255
linux@ubuntu:~/files$
```

In the above example, the gawk command prints all the line which matches the 'Sunil'.

- **To Split a line into fields:** For each line, the gawk command splits the record delimited by whitespace character by default and stores it in the \$n variables. If the line has 3 words, it will be stored in \$1, \$2 and \$3 respectively. \$0 represents the whole line.
gawk '{print \$2}' mobile.txt

```
linux@ubuntu:~/files$ gawk '{print $2}' mobile.txt
9984537565
8875546255
9865741259
8875463125
9822456788
linux@ubuntu:~/files$
```

In the above example, \$2 represents Monile no. field.

- **To display count of lines:**

```
gawk '{print NR, $0}' mobile.txt
```

```
linux@ubuntu:~/files$ gawk '{print NR,$0}' mobile.txt
1 Deepak 9984537565
2 Sunil 8875546255
3 Aman 9865741259
4 Kuldeep 8875463125
5 Sundip 9822456788
linux@ubuntu:~/files$
```

- **To find the length of the longest line present in the file:**

```
gawk '{ if (length($0) > max) max = length($0) } END { print max }'
mobile.txt
```

```
linux@ubuntu:~/files$ gawk '{ if(length($0) > max) max = length($0) } END {pr
18
linux@ubuntu:~/files$
```

- **To count the lines in a file:**

```
gawk 'END { print NR }' mobile.txt
```



```
linux@ubuntu:~/files$ gawk 'END { print NR }' mobile.txt
5
linux@ubuntu:~/files$
```

- To print lines with more than 5 characters:

```
gawk 'length($0) > 5' mobile.txt
```

```
linux@ubuntu:~/files$ gawk 'length($1) > 5' mobile.txt
Deepak 9984537565
Kuldeep 8875463125
Sundip 9822456788
linux@ubuntu:~/files$
```

Note:

- To check for the manual page of gawk command, use the following command:
man gawk
- To check the help page of gawk command, use the following command:
gawk --help

```
linux@ubuntu:~$ gawk --help
Usage: gawk [POSIX or GNU style options] -f progfile [--] file ...
Usage: gawk [POSIX or GNU style options] [--] 'program' file ...
POSIX options:          GNU long options: (standard)
    -f progfile          --file=progfile
    -F fs                --field-separator=fs
    -v var=val           --assign=var=val
Short options:          GNU long options: (extensions)
    -b                  --characters-as-bytes
    -c                  --traditional
    -C                  --copyright
    -d[file]            --dump-variables[=file]
    -D[file]            --debug[=file]
    -e 'program-text'   --source='program-text'
    -E file             --exec=file
    -g                  --gen-pot
    -h                  --help
    -i includefile      --include=includefile
    -l library          --load=library
    -L[fatal|invalid]   --lint[=fatal|invalid]
    -M                  --bignum
    -N                  --use-lc-numeric
    -n                  --non-decimal-data
    -o[file]            --pretty-print[=file]
    -O                  --optimize
    -p[file]            --profile[=file]
    -P                  --posix
    -r                  --re-interval
    -S                  --sandbox
    -t                  --lint-old
    -V                  --version

To report bugs, see node 'Bugs' in 'gawk.info', which is
section 'Reporting Problems and Bugs' in the printed version.

gawk is a pattern scanning and processing language.
By default it reads standard input and writes standard output.
```

Multiple Instructions(-e,-f)

There are several methods to specify multiple commands in a sed program.

Using newlines is most natural when running a sed script from a file (using the -f option).

On the command line, all sed commands may be separated by newlines. Alternatively, you may specify each command as an argument to an -e option

Context addresses

Context addresses are regular expressions enclosed in slashes `"/`. If you specify a context address for a command, **sed** only applies the editing function to those lines which match the regular expression. By using context addresses and a print function, you can improvise a **grep**-like behavior; for example, the shell script *mygrep*:

```
sed -n -e "/$1/p" <$2
```

This script uses the shell parameter **\$1** as a context address for **sed** to use in searching the file specified by the parameter **\$2**. Whenever **sed** finds a line matching the address given in **\$1**, it executes the **p** function (print) and outputs that line.

Note the **-n** argument to **sed** in this script; **sed** normally echoes every line it reads to its standard output. While the **-n** option is in effect, **sed** only prints when you tell it to with the **p** or **P** functions. Note also that if you want to use **sed** within a shell script and pass parameters to it, the **sed** instructions must be in *double* quotes, not single quotes.

Shell and Subshell

- **Shell and Subshell**

When the shell executes a shell script, it first spawns a subshell, which again executes commands in the script. When script execution is complete, the child returns control to the parent shell.

Ex. Subshell used to read the statements in if1.

```
sh $sh if1.sh
```

Thus a shell script run with sh need not have execute permission. This is applicable for executing only shell scripts.

Above ex. We specify script name from the shell prompt & uses a sub shell of the same type.

If the script contains the interpreter lines in this form.

```
#!/user/bin/ksh
```

Then, even though the login shell may be Bash, it will use the korn shell to execute the script.

Command Line Arguments in Linux Shell Scripting

Overview :

Command line arguments (also known as positional parameters) are the arguments specified at the command prompt with a command or script to be executed. The locations at the command prompt of the arguments as well as the location of the command, or the script itself, are stored in corresponding variables. These variables are special shell variables. Below picture will help you understand them.

Let's create a shell script with name "command_line_agruments.sh", it will show the command line arguments that were supplied and count number of arguments, value of first argument and Process ID (PID) of the Script.

```
linuxtechi@localhost:~$ cat command_line_agruments.sh
```

Assign Executable permissions to the Script

```
linuxtechi@localhost:~$ chmod +x command_line_agruments.sh
```

Now execute the scripts with command line arguments

```
linuxtechi@localhost:~$ ./command_line_agruments.sh Linux AIX HPUX VMware
```

```
There are 4 arguments specified at the command line.
```

```
The arguments supplied are: Linux AIX HPUX VMware
```

```
The first argument is: Linux
```

```
The PID of the script is: 16316
```

Shifting Command Line Arguments

The shift command is used to move command line arguments one position to the left. During this move, the first argument is lost. “command_line_agruments.sh” script below uses the shift command:

```
linuxtechi@localhost:~$ cat command_line_agruments.sh
```

Now Execute the Script again.

```
linuxtechi@localhost:~$ ./command_line_agruments.sh Linux AIX HPUX VMware
```

```
There are 4 arguments specified at the command line
```

```
The arguments supplied are: Linux AIX HPUX VMware
```

```
The first argument is: Linux
```

```
The Process ID of the script is: 16369
```

```
The new first argument after the first shift is: AIX
```

```
The new first argument after the second shift is: HPUX
```

```
linuxtechi@localhost:~$
```

Multiple shifts in a single attempt may be performed by furnishing the desired number of shifts to the shift command as an argument

Command Line Arguments in Shell

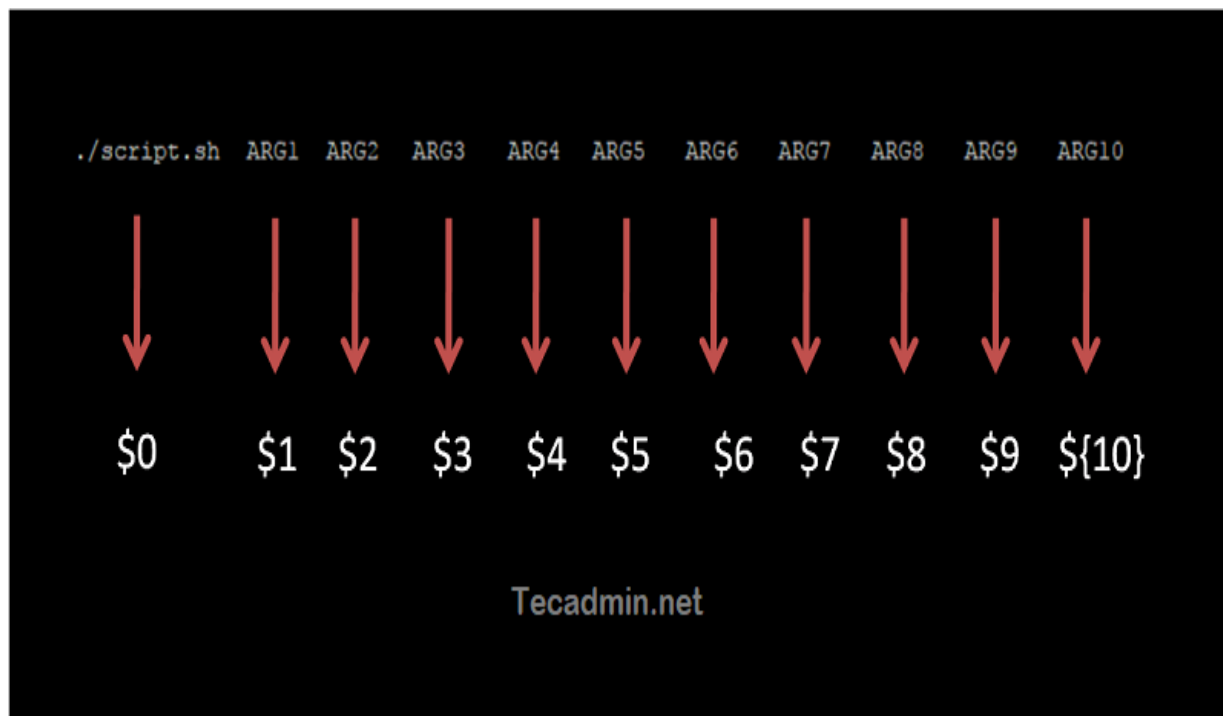
Script

Command line arguments are also known as positional parameters. These arguments are specific with the shell script on terminal during the run time. Each variable passed to a shell script at command line are stored in corresponding shell variables including the shell script name.

Syntax:

```
./myscript.sh ARG1 ARG2 ARG3 ARG4 ARG5 ARG6 ARG7 ARG8 ARG9 ARG10
```

See the below image to understand the command line values and variables. Here ARG1, ARG2 to ARG10 are command line values, which is assigned to corresponding shell variables.



These are also known as special variables provided by the shell. Except above screenshot, there are some more special variables as given below.

Special Variable Variable Details

`$1` to `$n`

`$1` is the first arguments, `$2` is second argument till `$n` n'th arguments. From 10'th argument, you must need to inclose them in braces like `${10}`, `${11}` and so on

`$0`

The name of script itself

`$$`

Process id of current shell

`$*`

Values of all the arguments. All arguments are double quoted

`$#`

Total number of arguments passed to script

`$@`

Values of all the arguments

\$?

Exit status id of last command

\$!

Process id of last command

Example Script

Command line arguments can be passed just after script file name with space separated. If any argument have space, put them under single or double quote. Read below simple script.

Shell



```
1 #!/bin/bash
2
3 ### Print total arguments and their values
4
5 echo "Total Arguments:" $#
6 echo "All Arguments values:" $@
7
8 ### Command arguments can be accessed as
9
10 echo "First->" $1
11 echo "Second->" $2
12
13 # You can also access all arguments in an array and use them in a script.
14
15 args=("$@")
16 echo "First->" ${args[0]}
17 echo "Second->" ${args[1]}
```

Now execute this script with 2 arguments and found following results.

```
$ ./arguments.sh Hello TecAdmin
```

```
root@tecadmin:~#
root@tecadmin:~#
root@tecadmin:~# ./arguments.sh Hello TecAdmin
Total Arguments: 2
All Arguments values: Hello TecAdmin
First-> Hello
Second-> TecAdmin
First-> Hello
Second-> TecAdmin
root@tecadmin:~#
root@tecadmin:~#
```

Exporting shell variables (export shell command)

A *local* shell variable is a variable known only to the shell that created it. If you start a new shell, the old shell's variables are unknown to it. If you want the new shells that you open to use the variables from an old shell, export the variables to make them *global*.

You can use the **export** command to make local variables global. To make your local shell variables global automatically, export them in your `.profile` file.

Note: Variables can be exported down to child shells but not exported up to parent shells.

See the following examples:

- To make the local shell variable `PATH` global, type the following:

```
export PATH
```

- To list all your exported variables, type the following:

```
export
```

The system displays information similar to the following:

NAME

export - Set export attribute for shell variables.

SYNOPSIS

```
export [-fn] [name[=value] ...] or export -p
```

DESCRIPTION

export- command is one of the bash shell BUILTINS commands, which means it is part of your shell. The export command is fairly simple to use as it has straightforward syntax with only three available command options. In general, the export command marks an environment variable to be exported with any newly forked child processes and thus it allows a child process to inherit all marked variables.

Options

Tag	Description
-p	List of all names that are exported in the current shell
-n	Remove names from export list
-f	Names are exported as functions

Example-1:

To set vim as a text editor

```
$ export EDITOR=/usr/bin/vim
```

output:

no output will be seen on screen , to see exported variable grep from exported ones.

```
$export|grepEDITOR
declare -x EDITOR="/usr/bin/vim"
```

Example-2:

To set colorful prompt

```
$ export PS1='\[\e[1;32m\]\[\u@\h \W\]\$[\e[0m\] '
```

output:

the colour of prompt will change to green.

Example-3:

To Set JAVA_HOME:

```
$ export JAVA_HOME=/usr/local/jdk
```

output:

no output will be seen on screen , to see exported variable grep from exported ones.

```
$ export | grep JAVA_HOME
```

```
declare -x JAVA_HOME="/usr/local/jdk"
```

Example-4:

To export shell function:

```
$ name () { echo "tutorialspoint"; }
```

```
$ export -f printname
```

output:

```
$name
```

```
tutorialspoint
```

Example-:

To remove names from exported list, use -n option

```
$ export -n EDITOR
```

note: in Example-3 we have set EDITOR=/usr/bin/vim, and have seen it in exported list.

output:

```
$ export | grep EDITOR
```

note: no output after grepping all exported variables, as EDITOR exported variable is removed from exported list.

Shell supports a different type of variable called an **array variable**. This can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

Defining Array Values

The difference between an array variable and a scalar variable can be explained as follows.

Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows –

```
NAME01="Zara"  
NAME02="Qadir"  
NAME03="Mahnaz"
```

```
NAME04="Ayan"
NAME05="Daisy"
```

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable. This helps assign a value to one of its indices.

```
array_name[index]=value
```

Here *array_name* is the name of the array, *index* is the index of the item in the array that you want to set, and *value* is the value you want to set for that item.

As an example, the following commands –

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

If you are using the **ksh** shell, here is the syntax of array initialization –

```
set -A array_name value1 value2 ... valuen
```

If you are using the **bash** shell, here is the syntax of array initialization –

```
array_name=(value1 ... valuen)
```

Accessing Array Values

After you have set any array variable, you access it as follows –

```
${array_name[index]}
```

Here *array_name* is the name of the array, and *index* is the index of the value to be accessed. Following is an example to understand the concept –

[Live Demo](#)

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

The above example will generate the following result –

```
$/test.sh
First Index: Zara
Second Index: Qadir
```

You can access all the items in an array in one of the following ways –

```
${array_name[*]}
${array_name[@]}
```

Here **array_name** is the name of the array you are interested in. Following example will help you understand the concept –

[Live Demo](#)

```
#!/bin/sh

NAME[0]="Zara"
```



```
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

The above example will generate the following result –

```
$/test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual tasks when needed.

Using functions to perform repetitive tasks is an excellent way to create **code reuse**. This is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions

To declare a function, simply use the following syntax –

```
function_name () {
    list of commands
}
```

The name of your function is **function_name**, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, followed by a list of commands enclosed within braces.

Example

Following example shows the use of function –

[Live Demo](#)

```
#!/bin/sh

# Define your function here
Hello () {
    echo "Hello World"
}

# Invoke your function
Hello
```

Upon execution, you will receive the following output –

```
$/test.sh
Hello World
```

Pass Parameters to a Function

You can define a function that will accept parameters while calling the function. These parameters would be represented by **\$1**, **\$2** and so on.

Following is an example where we pass two parameters *Zara* and *Ali* and then we capture and print these parameters in the function.

[Live Demo](#)

```
#!/bin/sh

# Define your function here
Hello () {
    echo "Hello World $1 $2"
}

# Invoke your function
Hello Zara Ali
```

Upon execution, you will receive the following result –

```
$/test.sh
Hello World Zara Ali
```

Returning Values from Functions

If you execute an **exit** command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the **return** command whose syntax is as follows –

return code

Here **code** can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example

Following function returns a value 10 –

[Live Demo](#)

```
#!/bin/sh

# Define your function here
Hello () {
    echo "Hello World $1 $2"
    return 10
}

# Invoke your function
Hello Zara Ali

# Capture value returned by last command
ret=$?

echo "Return value is $ret"
```

Upon execution, you will receive the following result –

```
$/test.sh
Hello World Zara Ali
Return value is 10
```

Nested Functions

One of the more interesting features of functions is that they can call themselves and also other functions. A function that calls itself is known as a **recursive function**.

Following example demonstrates nesting of two functions –

```
#!/bin/sh

# Calling one function from another
number_one () {
    echo "This is the first function speaking..."
    number_two
}

number_two () {
    echo "This is now the second function speaking..."
}

# Calling function one.
number_one
```

Upon execution, you will receive the following result –

```
This is the first function speaking...
This is now the second function speaking...
```

Function Call from Prompt

You can put definitions for commonly used functions inside your **.profile**. These definitions will be available whenever you log in and you can use them at the command prompt.

Alternatively, you can group the definitions in a file, say **test.sh**, and then execute the file in the current shell by typing –

```
$. test.sh
```

This has the effect of causing functions defined inside **test.sh** to be read and defined to the current shell as follows –

```
$ number_one
This is the first function speaking...
This is now the second function speaking...
$
```

To remove the definition of a function from the shell, use the **unset** command with the **.f** option. This command is also used to remove the definition of a variable to the shell.

```
$ unset -f function_name
```

UNIT-03

What is TCP/IP?

TCP/IP stands for Transmission Control Protocol/Internet Protocol and is a suite of communication protocols used to interconnect network devices on the internet. TCP/IP is also used as a communications protocol in a private computer network (an [intranet](#) or extranet).

The entire IP suite -- a set of rules and procedures -- is commonly referred to as TCP/IP. [TCP](#) and [IP](#) are the two main protocols, though others are included in the suite. The TCP/IP protocol suite functions as an abstraction layer between internet applications and the routing and switching fabric.

TCP/IP specifies how data is exchanged over the internet by providing end-to-end communications that identify how it should be broken into [packets](#), addressed, transmitted, routed and received at the destination. TCP/IP requires little central management and is designed to make networks reliable with the ability to recover automatically from the failure of any device on the network.

The two main protocols in the IP suite serve specific functions. TCP defines how applications can create channels of communication across a network. It also manages how a message is assembled into smaller packets before they are then transmitted over the internet and reassembled in the right order at the destination address.

IP defines how to address and route each packet to make sure it reaches the right destination. Each gateway computer on the network [checks this IP address](#) to determine where to forward the message.

A subnet mask tells a computer, or other network device, what portion of the IP address is used to represent the network and what part is used to represent hosts, or other computers, on the network.

Network address translation (NAT) is the virtualization of IP addresses. NAT helps improve security and decrease the number of IP addresses an organization needs.

Common TCP/IP protocols include the following:

Hypertext Transfer Protocol (HTTP) handles the communication between a web server and a web browser.

[HTTP Secure](#) handles secure communication between a web server and a web browser.

File Transfer Protocol handles transmission of files between computers.

How does TCP/IP work?

TCP/IP uses the [client-server model](#) of communication in which a user or machine (a client) is provided a service, like sending a webpage, by another computer (a server) in the network. Collectively, the TCP/IP suite of protocols is classified as [stateless](#), which means each client request is considered new because it is unrelated to previous requests. Being stateless frees up network paths so they can be used continuously.

The transport layer itself, however, is stateful. It transmits a single message, and its connection remains in place until all the packets in a message have been received and reassembled at the destination.

write command in Linux with Examples

- Last Updated : 15 Apr, 2019

write command in Linux is used to send a message to another user. The write utility allows a user to communicate with other users, by copying lines from one user's terminal to others. When you run the write command, the user you are writing to gets a message of the form:

Message from yourname@yourhost on yourtty at hh:mm ...

Any further lines the user enter will be copied to the specified user's terminal. If the other user wants to reply, they must run write as well. When you are done, type an end-of-file or interrupt character. The other user will see the message 'EOF' indicating that the conversation is over.

Syntax:

write user [tty]

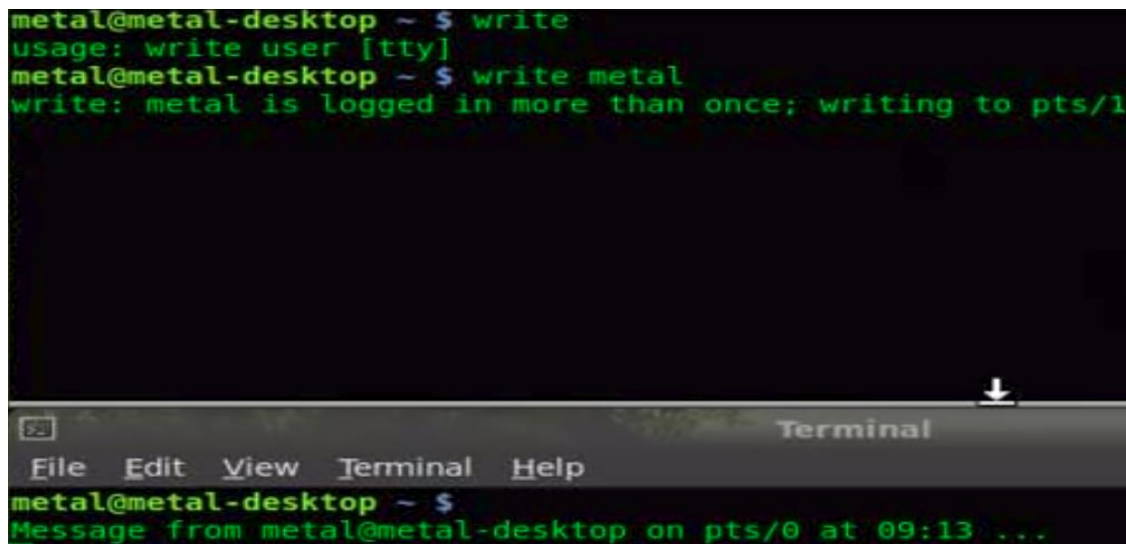
write command without any option: It will print the general syntax of the write. With the help of this, the user will get a generalized idea about how to use this command since there nothing like help option for the write command.

```
algoscale@algoscale-Lenovo-ideapad-330-15IKB:~$ write
usage: write user [tty]
```

Example:

- **write metal:** In this command *metal* is the name of another user that I'm logged in with and when I execute this command with a message I get a notification on my terminal showing that I received a message from another user.

write metal



```
metal@metal-desktop ~ $ write
usage: write user [tty]
metal@metal-desktop ~ $ write metal
write: metal is logged in more than once; writing to pts/1

metal@metal-desktop ~ $
Message from metal@metal-desktop on pts/0 at 09:13 ...
```

Explanation: So the basic use case of the *write* command is to send messages to the users on the other terminal as a way to interact. Once you enter write command and type your message then every user that is logged in will get a pop-up message. You will also receive a message from this particular user so if any other user wants to broadcast his message he can do the same.

The TCP/IP model differs slightly from the seven-layer Open Systems Interconnection (OSI) networking model designed after it. The OSI reference model defines how applications can communicate over a network.

Why is TCP/IP important?

TCP/IP is nonproprietary and, as a result, is not controlled by any single company.

Therefore, the IP suite can be modified easily. It is compatible with all operating systems

(OSes), so it can communicate with any other system. The IP suite is also compatible with all types of computer hardware and networks.

TCP/IP is highly scalable and, as a routable protocol, can determine the most efficient path through the network. It is widely used in current internet architecture.

The 4 layers of the TCP/IP model

TCP/IP functionality is divided into four layers, each of which includes specific protocols: The [application layer](#) provides applications with standardized data exchange. Its protocols include HTTP, FTP, [Post Office Protocol 3](#), [Simple Mail Transfer Protocol](#) and Simple Network Management Protocol. At the application layer, the payload is the actual application data.

The transport layer is responsible for maintaining end-to-end communications across the network. TCP handles communications between hosts and provides flow control, multiplexing and reliability. The transport protocols include TCP and [User Datagram Protocol](#), which is sometimes used instead of TCP for special purposes.

The network layer, also called the *internet layer*, deals with packets and connects independent networks to transport the packets across network boundaries. The network layer protocols are IP and Internet Control Message Protocol, which is used for error reporting.

The physical layer, also known as the *network interface layer* or *data link layer*, consists of protocols that operate only on a link -- the network component that interconnects nodes or hosts in the network. The protocols in this lowest layer include Ethernet for local area networks and [Address Resolution Protocol](#).

Uses of TCP/IP

TCP/IP can be used to provide remote login over the network for interactive file transfer to deliver email, to deliver webpages over the network and to remotely access a server host's file system. Most broadly, it is used to represent how information changes form as it travels over a network from the concrete physical layer to the abstract application layer. It details the basic protocols, or methods of communication, at each layer as information passes through.

Pros and cons of TCP/IP

The advantages of using the TCP/IP model include the following:

- helps establish a connection between different types of computers;
- works independently of the OS;
- supports many routing protocols;
- uses client-server architecture that is highly scalable;
- can be operated independently;
- supports several routing protocols; and
- is lightweight and doesn't place unnecessary strain on a network or computer.

The disadvantages of TCP/IP include the following:

- is complicated to set up and manage;
- transport layer does not guarantee delivery of packets;
- is not easy to replace protocols in TCP/IP;
- does not clearly separate the concepts of services, interfaces and protocols, so it is not suitable for describing new technologies in new networks; and
- is especially vulnerable to a [synchronization attack](#), which is a type of denial-of-service attack in which a bad actor uses TCP/IP.

How are TCP/IP and IP different?

There are numerous differences between TCP/IP and IP. For example, IP is a low-level internet protocol that facilitates data communications over the internet. Its purpose is to

deliver [packets of data that consist of a header](#), which contains routing information, such as source and destination of the data, and the data payload itself.

IP is limited by the amount of data that it can send. The maximum size of a single IP data packet, which contains both the header and the data, is between 20 and 24 bytes long. This means that longer strings of data must be broken into multiple data packets that must be independently sent and then reorganized into the correct order after they are sent.

Since IP is strictly a data send/receive protocol, there is no built-in checking that verifies whether the data packets sent were actually received.

In contrast to IP, TCP/IP is a higher-level smart communications protocol that can do more things. TCP/IP still uses IP as a means of transporting data packets, but it also connects computers, applications, webpages and web servers. TCP understands holistically the entire streams of data that these assets require in order to operate, and it makes sure the entire volume of data needed is sent the first time. TCP also runs checks that ensure the data is delivered.

As it does its work, TCP can also control the size and flow rate of data. It ensures that networks are free of any congestion that could block the receipt of data.

An example is an application that wants to send a large amount of data over the internet. If the application only used IP, the data would have to be broken into multiple IP packets. This would require multiple requests to send and receive data, since IP requests are issued per packet.

With TCP, only a single request to send an entire data stream is needed; TCP handles the rest. Unlike IP, TCP can detect problems that arise in IP and request retransmission of any data packets that were lost. TCP can also reorganize packets so they get transmitted in the proper order -- and it can minimize network congestion. TCP/IP makes data transfers over the internet easier.

The history of TCP/IP

The Defense Advanced Research Projects Agency, the research branch of the U.S.

Department of Defense, created the TCP/IP model in the 1970s for use in ARPANET, a wide area network that preceded the internet. TCP/IP was originally designed for the Unix OS, and it has been built into all of the OSes that came after it.

The TCP/IP model and its related protocols are now maintained by the Internet Engineering Task Force.

Firewall History

Firewalls have existed since the late 1980's and started out as packet filters, which were networks set up to examine packets, or bytes, transferred between computers. Though packet filtering firewalls are still in use today, firewalls have come a long way as technology has developed throughout the decades.

Gen 1 Virus

Generation 1, Late 1980's, virus attacks on stand-alone PC's affected all businesses and drove anti-virus products.

Gen 2 Networks

Generation 2, Mid 1990's, attacks from the internet affected all business and drove creation of the firewall.

Gen 3 Applications

Generation 3, Early 2000's, exploiting vulnerabilities in applications which affected most businesses and drove Intrusion Prevention Systems Products (IPS).

Gen 4 Payload

Generation 4, Approx. 2010, rise of targeted, unknown, evasive, polymorphic attacks which affected most businesses and drove anti-bot and sandboxing products.

Gen 5 Mega

Generation 5, Approx. 2017, large scale, multi-vector, mega attacks using advance attack tools and is driving advance threat prevention solutions.

Back in 1993, Check Point CEO Gil Shwed introduced the first stateful inspection firewall, FireWall-1. Fast forward twenty-seven years, and a firewall is still an organization's first line of defense against cyber attacks. Today's firewalls, including [Next Generation Firewalls and Network Firewalls](#) support a wide variety of functions and capabilities with built-in features, including:

[Network Threat Prevention](#)

[Application and Identity-Based Control](#)

[Hybrid Cloud Support](#)

[Scalable Performance](#)

Types of Firewalls

Packet filtering

A small amount of data is analyzed and distributed according to the filter's standards.

Proxy service

Network security system that protects while filtering messages at the application layer.

Stateful inspection

Dynamic packet filtering that monitors active connections to determine which network packets to allow through the Firewall.

Next Generation Firewall (NGFW)

Deep packet inspection Firewall with application-level inspection.

What Firewalls Do?

A Firewall is a necessary part of any security architecture and takes the guesswork out of host level protections and entrusts them to your network security device. Firewalls, and especially Next Generation Firewalls, focus on blocking malware and application-layer attacks, along with an integrated intrusion prevention system (IPS), these Next Generation Firewalls can react quickly and seamlessly to detect and react to outside attacks across the whole network. They can set policies to better defend your network and carry out quick assessments to detect invasive or suspicious activity, like malware, and shut it down.

Why Do We Need Firewalls?

Firewalls, especially [Next Generation Firewalls](#), focus on blocking malware and application-layer attacks. Along with an [integrated intrusion prevention system \(IPS\)](#), these Next Generation Firewalls are able to react quickly and seamlessly to detect and combat attacks across the whole network. Firewalls can act on previously set policies to better protect your network and can carry out quick assessments to detect invasive or suspicious activity, such as malware, and shut it

down. By leveraging a firewall for your security infrastructure, you're setting up your network with specific policies to allow or block incoming and outgoing traffic.

Network Layer vs. Application Layer Inspection

Network layer or packet filters inspect packets at a relatively low level of the TCP/IP protocol stack, not allowing packets to pass through the firewall unless they match the established rule set where the source and destination of the rule set is based upon Internet Protocol (IP) addresses and ports. Firewalls that do network layer inspection perform better than similar devices that do application layer inspection. The downside is that unwanted applications or malware can pass over allowed ports, e.g. outbound Internet traffic over web protocols HTTP and HTTPS, port 80 and 443 respectively.

The Importance of NAT and VPN

Firewalls also perform basic network level functions such as Network Address Translation (NAT) and Virtual Private Network (VPN). Network Address Translation hides or translates internal client or server IP addresses that may be in a "private address range", as defined in RFC 1918 to a public IP address. Hiding the addresses of protected devices preserves the limited number of IPv4 addresses and is a defense against network reconnaissance since the IP address is hidden from the Internet.

Similarly, a [virtual private network \(VPN\)](#) extends a private network across a public network within a tunnel that is often encrypted where the contents of the packets are protected while traversing the Internet. This enables users to safely send and receive data across shared or public networks.

Next Generation Firewalls and Beyond

Next Generation Firewalls inspect packets at the application level of the TCP/IP stack and are able to identify applications such as Skype, or Facebook and enforce security policy based upon the type of application.

Today, UTM (Unified Threat Management) devices and Next Generation Firewalls also include threat prevention technologies such as [intrusion prevention system \(IPS\)](#) or [Antivirus](#) to detect and prevent malware and threats. These devices may also include sandboxing technologies to detect threats in files.

As the cyber security landscape continues to evolve and attacks become more sophisticated, Next Generation Firewalls will continue to be an essential component of any organization's security solution, whether you're in the data center, network, or cloud. To learn more about the essential capabilities your Next Generation Firewall needs to have, download the [Next Generation Firewall \(NGFW\) Buyer's Guide](#) today

What is a Firewall?

A Firewall is a network security device that monitors and filters incoming and outgoing network traffic based on an organization's previously established security

policies. At its most basic, a firewall is essentially the barrier that sits between a private internal network and the public Internet. A firewall's main purpose is to allow non-threatening traffic in and to keep dangerous traffic out

write command in Linux with Examples

- Last Updated : 15 Apr, 2019

write command in Linux is used to send a message to another user. The write utility allows a user to communicate with other users, by copying lines from one user's terminal to others. When you run the write command, the user you are writing to gets a message of the form:

Message from yourname@yourhost on yourtty at hh:mm ...

Any further lines the user enter will be copied to the specified user's terminal. If the other user wants to reply, they must run write as well. When you are done, type an end-of-file or interrupt character. The other user will see the message 'EOF' indicating that the conversation is over.

Syntax:

write user [tty]

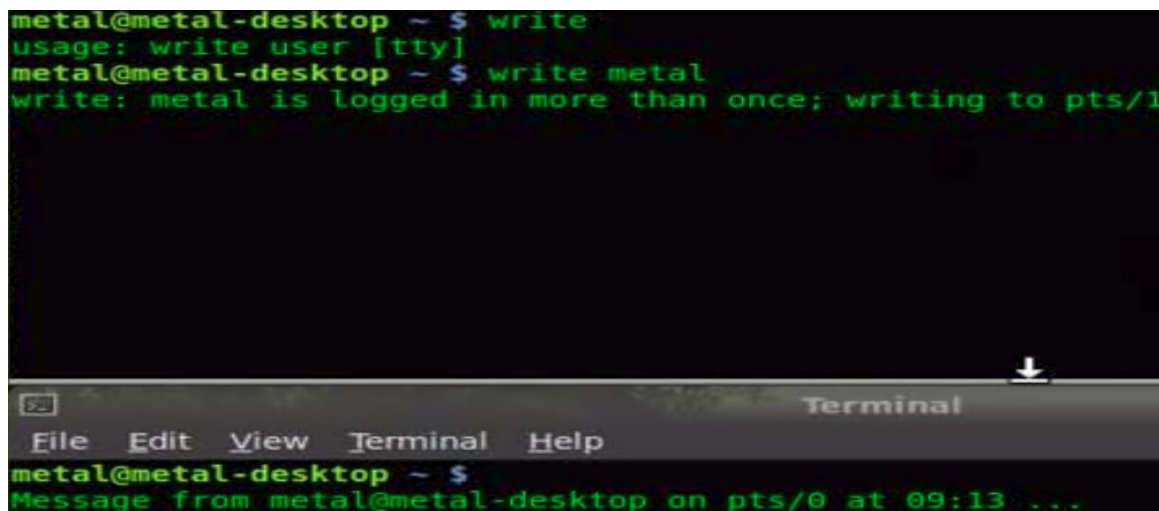
write command without any option: It will print the general syntax of the write. With the help of this, the user will get a generalized idea about how to use this command since there nothing like help option for the write command.

```
algoscale@algoscale-Lenovo-Ideapad-330-15IKB:~$ write
usage: write user [tty]
```

Example:

- **write metal:** In this command *metal* is the name of another user that I'm logged in with and when I execute this command with a message I get a notification on my terminal showing that I received a message from another user.

write metal



```
metal@metal-desktop ~ $ write
usage: write user [tty]
metal@metal-desktop ~ $ write metal
write: metal is logged in more than once; writing to pts/1

Terminal
File Edit View Terminal Help
metal@metal-desktop ~ $
Message from metal@metal-desktop on pts/0 at 09:13 ...
```

Explanation: So the basic use case of the *write* command is to send messages to the users on the other terminal as a way to interact. Once you enter write command and type your message then every user that is logged in will get a pop-up message. You will also receive a message from this particular user so if any other user wants to broadcast his message he can do the same.

wall command in Linux with Examples

- Last Updated : 27 May, 2019

wall command in Linux system is used to write a message to all users. This command displays a message, or the contents of a file, or otherwise its standard input, on the terminals of all currently logged in users. The lines which will be longer than 79 characters, wrapped by this command. Short lines are whitespace padded to have 79 characters. A carriage return and newline at the end of each line is put by *wall* command always. Only the superuser can write on the terminals of users who have chosen to deny messages or are using a program which automatically denies messages. Reading from a file is refused when the invoker is not superuser and the program is *suid(set-user-ID)* or *sgid(set-group-ID)*.

Syntax:

```
wall [-n] [-t timeout] [message | file]
```

Options:

- **wall -n:** This option will suppress the banner.
`wall -n`
- **wall -t:** This option will abandon the write attempt to the terminals after timeout seconds. This timeout needs to be a positive integer. The by default value is 300 seconds, which is a legacy from the time when peoples ran terminals over modem lines.

Example:

```
wall -t 30
```

- **wall -V :** This option display version information and exit.
`wall -V`

```
algoscale@algoscale-Lenovo-ideapad-330-15IKB:~$ wall -V
wall from util-linux 2.27.1
```

- **wall -h :** This option will display help message and exit.
`wall -h`

```
algoscale@algoscale-Lenovo-ideapad-330-15IKB:~$ wall -h

Usage:
wall [options] [<file> | <message>]

Write a message to all users.

Options:
-n, --nobanner          do not print banner, works only for root
-t, --timeout <timeout> write timeout in seconds

-h, --help              display this help and exit
-V, --version           output version information and exit

For more details see wall(1).
```

What is TCP/IP?

TCP/IP stands for Transmission Control Protocol/Internet Protocol and is a suite of communication protocols used to interconnect network devices on the internet. TCP/IP is also used as a communications protocol in a private computer network (an [intranet](#) or extranet).

The entire IP suite -- a set of rules and procedures -- is commonly referred to as TCP/IP. [TCP](#) and [IP](#) are the two main protocols, though others are included in the suite. The TCP/IP protocol suite functions as an abstraction layer between internet applications and the routing and switching fabric.

TCP/IP specifies how data is exchanged over the internet by providing end-to-end communications that identify how it should be broken into [packets](#), addressed, transmitted, routed and received at the destination. TCP/IP requires little central management and is designed to make networks reliable with the ability to recover automatically from the failure of any device on the network.

The two main protocols in the IP suite serve specific functions. TCP defines how applications can create channels of communication across a network. It also manages how a message is assembled into smaller packets before they are then transmitted over the internet and reassembled in the right order at the destination address.

IP defines how to address and route each packet to make sure it reaches the right destination. Each gateway computer on the network [checks this IP address](#) to determine where to forward the message.

A subnet mask tells a computer, or other network device, what portion of the IP address is used to represent the network and what part is used to represent hosts, or other computers, on the network.

Network address translation (NAT) is the virtualization of IP addresses. NAT helps improve security and decrease the number of IP addresses an organization needs.

Common TCP/IP protocols include the following:

Hypertext Transfer Protocol (HTTP) handles the communication between a web server and a web browser.

[HTTP Secure](#) handles secure communication between a web server and a web browser.

File Transfer Protocol handles transmission of files between computers.

How does TCP/IP work?

TCP/IP uses the [client-server model](#) of communication in which a user or machine (a client) is provided a service, like sending a webpage, by another computer (a server) in the network. Collectively, the TCP/IP suite of protocols is classified as [stateless](#), which means each client request is considered new because it is unrelated to previous requests. Being stateless frees up network paths so they can be used continuously.

The transport layer itself, however, is stateful. It transmits a single message, and its connection remains in place until all the packets in a message have been received and reassembled at the destination.

write command in Linux with Examples

- Last Updated : 15 Apr, 2019

write command in Linux is used to send a message to another user. The write utility allows a user to communicate with other users, by copying lines from one user's terminal to others. When you run the write command, the user you are writing to gets a message of the form:

Message from yourname@yourhost on yourtty at hh:mm ...

Any further lines the user enters will be copied to the specified user's terminal. If the other user wants to reply, they must run `write` as well. When you are done, type an end-of-file or interrupt character. The other user will see the message 'EOF' indicating that the conversation is over.

Syntax:

`write user [tty]`

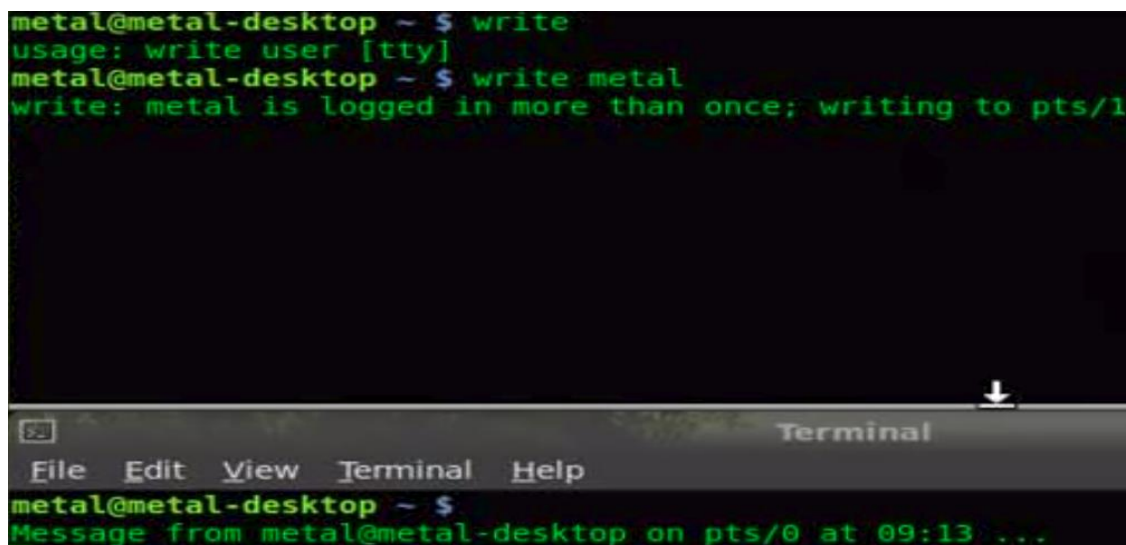
write command without any option: It will print the general syntax of the `write` command. With the help of this, the user will get a generalized idea about how to use this command since there is nothing like a `help` option for the `write` command.

```
algoscale@algoscale-Lenovo-ideapad-330-15IKB:~$ write
usage: write user [tty]
```

Example:

- **write metal:** In this command *metal* is the name of another user that I'm logged in with and when I execute this command with a message I get a notification on my terminal showing that I received a message from another user.

`write metal`



```
metal@metal-desktop ~ $ write
usage: write user [tty]
metal@metal-desktop ~ $ write metal
write: metal is logged in more than once; writing to pts/1

metal@metal-desktop ~ $
Message from metal@metal-desktop on pts/0 at 09:13 ...
```

Explanation: So the basic use case of the `write` command is to send messages to the users on the other terminal as a way to interact. Once you enter `write` command and type your message then every user that is logged in will get a pop-up message. You will also receive a message from this particular user so if any other user wants to broadcast his message he can do the same.

The TCP/IP model differs slightly from the seven-layer Open Systems Interconnection (OSI) networking model designed after it. The OSI reference model defines how applications can communicate over a network.

Why is TCP/IP important?

TCP/IP is nonproprietary and, as a result, is not controlled by any single company.

Therefore, the IP suite can be modified easily. It is compatible with all operating systems (OSes), so it can communicate with any other system. The IP suite is also compatible with all types of computer hardware and networks.

TCP/IP is highly scalable and, as a routable protocol, can determine the most efficient path through the network. It is widely used in current internet architecture.

The 4 layers of the TCP/IP model

TCP/IP functionality is divided into four layers, each of which includes specific protocols:

The [application layer](#) provides applications with standardized data exchange. Its protocols include HTTP, FTP, [Post Office Protocol 3](#), [Simple Mail Transfer Protocol](#) and Simple Network Management Protocol. At the application layer, the payload is the actual application data.

The transport layer is responsible for maintaining end-to-end communications across the network. TCP handles communications between hosts and provides flow control, multiplexing and reliability. The transport protocols include TCP and [User Datagram Protocol](#), which is sometimes used instead of TCP for special purposes.

The network layer, also called the *internet layer*, deals with packets and connects independent networks to transport the packets across network boundaries. The network layer protocols are IP and Internet Control Message Protocol, which is used for error reporting.

The physical layer, also known as the *network interface layer* or *data link layer*, consists of protocols that operate only on a link -- the network component that interconnects nodes or hosts in the network. The protocols in this lowest layer include Ethernet for local area networks and [Address Resolution Protocol](#).

Uses of TCP/IP

TCP/IP can be used to provide remote login over the network for interactive file transfer to deliver email, to deliver webpages over the network and to remotely access a server host's file system. Most broadly, it is used to represent how information changes form as it travels over a network from the concrete physical layer to the abstract application layer. It details the basic protocols, or methods of communication, at each layer as information passes through.

Pros and cons of TCP/IP

The advantages of using the TCP/IP model include the following:

- helps establish a connection between different types of computers;
- works independently of the OS;
- supports many routing protocols;
- uses client-server architecture that is highly scalable;
- can be operated independently;
- supports several routing protocols; and
- is lightweight and doesn't place unnecessary strain on a network or computer.

The disadvantages of TCP/IP include the following:

- is complicated to set up and manage;
- transport layer does not guarantee delivery of packets;
- is not easy to replace protocols in TCP/IP;
- does not clearly separate the concepts of services, interfaces and protocols, so it is not suitable for describing new technologies in new networks; and
- is especially vulnerable to a [synchronization attack](#), which is a type of denial-of-service attack in which a bad actor uses TCP/IP.

How are TCP/IP and IP different?

There are numerous differences between TCP/IP and IP. For example, IP is a low-level internet protocol that facilitates data communications over the internet. Its purpose is to deliver [packets of data that consist of a header](#), which contains routing information, such as source and destination of the data, and the data payload itself.

IP is limited by the amount of data that it can send. The maximum size of a single IP data packet, which contains both the header and the data, is between 20 and 24 bytes long. This means that longer strings of data must be broken into multiple data packets that must be independently sent and then reorganized into the correct order after they are sent.

Since IP is strictly a data send/receive protocol, there is no built-in checking that verifies whether the data packets sent were actually received.

In contrast to IP, TCP/IP is a higher-level smart communications protocol that can do more things. TCP/IP still uses IP as a means of transporting data packets, but it also connects computers, applications, webpages and web servers. TCP understands holistically the entire streams of data that these assets require in order to operate, and it makes sure the entire volume of data needed is sent the first time. TCP also runs checks that ensure the data is delivered.

As it does its work, TCP can also control the size and flow rate of data. It ensures that networks are free of any congestion that could block the receipt of data.

An example is an application that wants to send a large amount of data over the internet. If the application only used IP, the data would have to be broken into multiple IP packets. This would require multiple requests to send and receive data, since IP requests are issued per packet.

With TCP, only a single request to send an entire data stream is needed; TCP handles the rest. Unlike IP, TCP can detect problems that arise in IP and request retransmission of any data packets that were lost. TCP can also reorganize packets so they get transmitted in the proper order -- and it can minimize network congestion. TCP/IP makes data transfers over the internet easier.

The history of TCP/IP

The Defense Advanced Research Projects Agency, the research branch of the U.S. Department of Defense, created the TCP/IP model in the 1970s for use in ARPANET, a wide area network that preceded the internet. TCP/IP was originally designed for the Unix OS, and it has been built into all of the OSes that came after it.

The TCP/IP model and its related protocols are now maintained by the Internet Engineering Task Force.

Firewall History

Firewalls have existed since the late 1980's and started out as packet filters, which were networks set up to examine packets, or bytes, transferred between computers. Though packet filtering firewalls are still in use today, firewalls have come a long way as technology has developed throughout the decades.

Gen 1 Virus

Generation 1, Late 1980's, virus attacks on stand-alone PC's affected all businesses and drove anti-virus products.

Gen 2 Networks

Generation 2, Mid 1990's, attacks from the internet affected all business and drove creation of the firewall.

Gen 3 Applications

Generation 3, Early 2000's, exploiting vulnerabilities in applications which affected most businesses and drove Intrusion Prevention Systems Products (IPS).

Gen 4 Payload

Generation 4, Approx. 2010, rise of targeted, unknown, evasive, polymorphic attacks which affected most businesses and drove anti-bot and sandboxing products.

Gen 5 Mega

Generation 5, Approx. 2017, large scale, multi-vector, mega attacks using advance attack tools and is driving advance threat prevention solutions.

Back in 1993, Check Point CEO Gil Shwed introduced the first stateful inspection firewall, FireWall-1. Fast forward twenty-seven years, and a firewall is still an organization's first line of defense against cyber attacks. Today's firewalls, including [Next Generation Firewalls and Network Firewalls](#) support a wide variety of functions and capabilities with built-in features, including:

[Network Threat Prevention](#)

[Application and Identity-Based Control](#)

[Hybrid Cloud Support](#)

[Scalable Performance](#)

Types of Firewalls

Packet filtering

A small amount of data is analyzed and distributed according to the filter's standards.

Proxy service

Network security system that protects while filtering messages at the application layer.

Stateful inspection

Dynamic packet filtering that monitors active connections to determine which network packets to allow through the Firewall.

Next Generation Firewall (NGFW)

Deep packet inspection Firewall with application-level inspection.

What Firewalls Do?

A Firewall is a necessary part of any security architecture and takes the guesswork out of host level protections and entrusts them to your network security device. Firewalls, and especially Next Generation Firewalls, focus on blocking malware and application-layer attacks, along with an integrated intrusion prevention system (IPS), these Next Generation Firewalls can react quickly and seamlessly to detect and react to outside attacks across the whole network. They can set policies to better defend your network and carry out quick assessments to detect invasive or suspicious activity, like malware, and shut it down.

Why Do We Need Firewalls?

Firewalls, especially [Next Generation Firewalls](#), focus on blocking malware and application-layer attacks. Along with an [integrated intrusion prevention system \(IPS\)](#), these Next Generation Firewalls are able to react quickly and seamlessly to detect and combat attacks across the whole network. Firewalls can act on previously set policies to better protect your network and can carry out quick assessments to detect invasive or suspicious activity, such as malware, and shut it down. By leveraging a firewall for your security infrastructure, you're setting up your network with specific policies to allow or block incoming and outgoing traffic.

Network Layer vs. Application Layer Inspection

Network layer or packet filters inspect packets at a relatively low level of the TCP/IP protocol stack, not allowing packets to pass through the firewall unless they match the established rule set where the source and destination of the rule set is based upon Internet Protocol (IP) addresses and ports. Firewalls that do network layer inspection perform better than similar devices that do application layer inspection. The downside is that unwanted applications or malware can pass over allowed ports, e.g. outbound Internet traffic over web protocols HTTP and HTTPS, port 80 and 443 respectively.

The Importance of NAT and VPN

Firewalls also perform basic network level functions such as Network Address Translation (NAT) and Virtual Private Network (VPN). Network Address Translation hides or translates internal client or server IP addresses that may be in a “private address range”, as defined in RFC 1918 to a public IP address. Hiding the addresses of protected devices preserves the limited number of IPv4 addresses and is a defense against network reconnaissance since the IP address is hidden from the Internet.

Similarly, a [virtual private network \(VPN\)](#) extends a private network across a public network within a tunnel that is often encrypted where the contents of the packets are protected while traversing the Internet. This enables users to safely send and receive data across shared or public networks.

Next Generation Firewalls and Beyond

Next Generation Firewalls inspect packets at the application level of the TCP/IP stack and are able to identify applications such as Skype, or Facebook and enforce security policy based upon the type of application.

Today, UTM (Unified Threat Management) devices and Next Generation Firewalls also include threat prevention technologies such as [intrusion prevention system \(IPS\)](#) or [Antivirus](#) to detect and prevent malware and threats. These devices may also include sandboxing technologies to detect threats in files.

As the cyber security landscape continues to evolve and attacks become more sophisticated, Next Generation Firewalls will continue to be an essential component of any organization’s security solution, whether you’re in the data center, network, or cloud. To learn more about the essential capabilities your Next Generation Firewall needs to have, download the [Next Generation Firewall \(NGFW\) Buyer’s Guide](#) today

What is a Firewall?

A Firewall is a network security device that monitors and filters incoming and outgoing network traffic based on an organization’s previously established security policies. At its most basic, a firewall is essentially the barrier that sits between a private internal network and the public Internet. A firewall’s main purpose is to allow non-threatening traffic in and to keep dangerous traffic out

write command in Linux with Examples

- Last Updated : 15 Apr, 2019

write command in Linux is used to send a message to another user. The write utility allows a user to communicate with other users, by copying lines from one user's terminal to others. When you run the write command, the user you are writing to gets a message of the form:

Message from yourname@yourhost on yourtty at hh:mm ...

Any further lines the user enter will be copied to the specified user's terminal. If the other user wants to reply, they must run write as well. When you are done, type an end-of-file or interrupt character. The other user will see the message 'EOF' indicating that the conversation is over.

Syntax:

write user [tty]

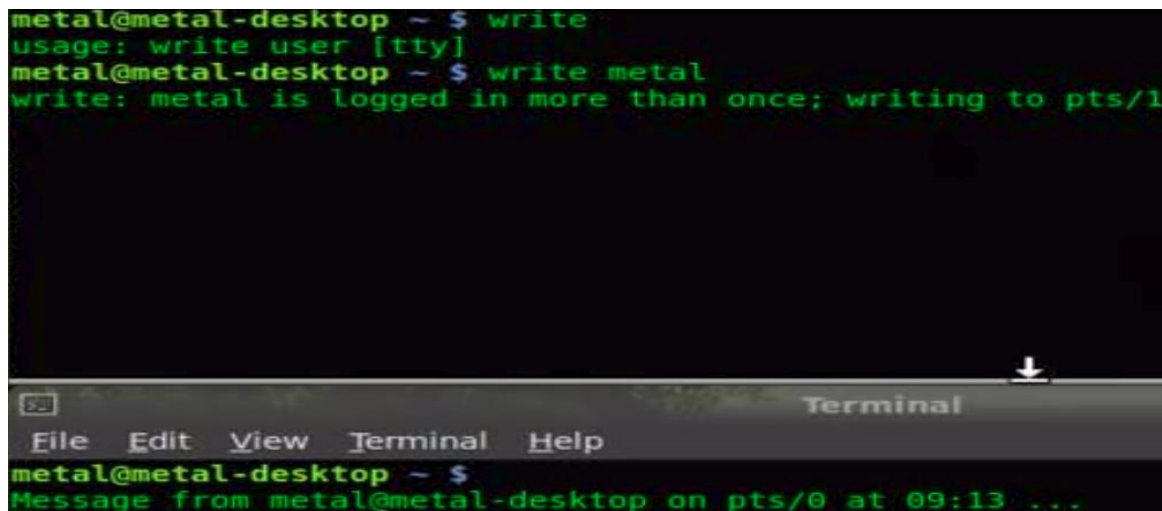
write command without any option: It will print the general syntax of the write. With the help of this, the user will get a generalized idea about how to use this command since there nothing like help option for the write command.

```
algoscale@algoscale-Lenovo-ideapad-330-15IKB:~$ write
usage: write user [tty]
```

Example:

- **write metal:** In this command *metal* is the name of another user that I'm logged in with and when I execute this command with a message I get a notification on my terminal showing that I received a message from another user.

write metal



```
metal@metal-desktop ~ $ write
usage: write user [tty]
metal@metal-desktop ~ $ write metal
write: metal is logged in more than once; writing to pts/1
Message from metal@metal-desktop on pts/0 at 09:13 ...
```

Explanation: So the basic use case of the *write* command is to send messages to the users on the other terminal as a way to interact. Once you enter write command and type your message then every user that is logged in will get a pop-up message. You will also receive a message from this particular user so if any other user wants to broadcast his message he can do the same.

wall command in Linux with Examples

- Last Updated : 27 May, 2019

wall command in Linux system is used to write a message to all users. This command displays a message, or the contents of a file, or otherwise its standard input, on the terminals of all currently logged in users. The lines which will be longer than 79 characters, wrapped by this command. Short lines are whitespace padded to have 79 characters. A carriage return and newline at the end of each line is put by *wall* command always. Only the superuser can write on the terminals of users who have chosen to deny messages or are using a program which

automatically denies messages. Reading from a file is refused when the invoker is not superuser and the program is *suid(set-user-ID)* or *sgid(set-group-ID)*.

Syntax:

```
wall [-n] [-t timeout] [message | file]
```

Options:

- **wall -n:** This option will suppress the banner.
`wall -n`
- **wall -t:** This option will abandon the write attempt to the terminals after timeout seconds. This timeout needs to be a positive integer. The by default value is 300 seconds, which is a legacy from the time when peoples ran terminals over modem lines.

Example:

```
wall -t 30
```

- **wall -V :** This option display version information and exit.

```
wall -V
```

```
algoscale@algoscale-Lenovo-ideapad-330-15IKB:~$ wall -V
wall from util-linux 2.27.1
```

- **wall -h :** This option will display help message and exit.

```
wall -h
```

```
algoscale@algoscale-Lenovo-ideapad-330-15IKB:~$ wall -h

Usage:
 wall [options] [<file> | <message>]

Write a message to all users.

Options:
 -n, --nobanner          do not print banner, works only for root
 -t, --timeout <timeout> write timeout in seconds

 -h, --help             display this help and exit
 -V, --version          output version information and exit

For more details see wall(1).
```