Example questions: file system statistics using the df command

The **df** command displays information about space usage on the file systems.

NOTE: THERE MAY BE A DELAY AS YOUR FILE SYSTEMS ARE BEING EXAMINED!

Type **df** to see information on the various mounted file systems on your computer.

Type **df .** to show information on the file system for the current directory. A response similar to the following will be seen.

Filesystem	**1K-blocks**	**Used**	**Available**	**Use%**	**Mounted on**
/dev/sd3a	**58502432**	**12749248**	**45753184**	**22%**	**/**

The report shows the file system (disk partition) sizes in **1kByte** blocks, with column 2 showing the full size, column 3 showing the Used space and column 4 showing the Available space. Column 5 shows the percentage space that is used.

Type **df -h .** to show the space sizes in human readable form, e.g. megabytes, gigabytes etc. The response will be similar to the following:

Filesystem	**Size**	**Used**	**Avail**	**Use%**	**Mounted on**
/dev/sd3a	**56G**	**13G**	**44G**	**22%**	**/**

Questions:

Provide a single line command answer to the following:

Q. Find and display the size of the file system, in **disk blocks**, where your current directory resides. Your answer will be returned to a variable called **disk_sizeB**.

A solution:
disk_sizeB=$(df . | awk ' { print $2 } ' | tail –1) ; echo $disk_sizeB

Q. Find and display the size of the file system, in disk **human readable** form, where your current directory resides. Your answer will be returned to a variable called **disk_sizeH**.

A solution:
disk_sizeH=$(df -h . | awk ' { print $2 } ' | tail –1) ; echo $disk_sizeH

Q. Find and display the percentage of space used for the file system, where your current directory resides. Your answer will be returned to a variable called **usage**.

A solution:
usage=$(df -h . | awk ' { print $5 } ' | tail –1) ; echo $usage

Example problem

Write a utility that will check the amount of disk space that is available on your disk, and if there is more than 90% of the disk space in use, then send a warning message to the user to advise that the disk is more than 90% full.

A solution

In the example above, we saw how to get a variable, e.g. **usage**, to represent the percentage of space used on the disk, as follows:

usage=$(df -h . | awk ' { print $5 } ' | tail –1)

The problem is that the variable is the form **num%**, but we need a simple integer variable. The following line uses the **sed** utilty (see lab notes) to substitute the % character with a blank space character:

usageNum=$(echo $usage | sed ' s/%/ / ')

The variable **usageNum** now represents an integer value (e.g. 22) to denote the percentage of used disk space.

Now we can write our script program, as follows:

```
#! /bin/bash
# Program to check disk space usage against a specific % limit
# DH  29/March/2007

#Check percentage usage
usage=$( df -h . | awk ' { print $5 } ' | tail -1 )

# Use sed to delete the % character
usageNum=$( echo $usage | sed ' s/%/ /' )

# Check usage against the limit (90%)
if (( usageNum >= 90 )); then
echo "WARNING: your disk is more than 90% full!!!"
else echo "OK - Your disk is not more than 90% full!"
fi

exit
```

NOTE: In the above example, the following two commands could have been piped together, but there were used separately for clarity of teaching and reading of the program:

```
THESE TWO COMMANDS:
usage=$( df -h . | awk ' { print $5 } ' | tail -1 )
usageNum=$( echo $usage | sed ' s/%/ /' )
CAN BE COMBINED AS FOLLOWS:

usageNum=$( df -h . | awk ' { print $5 } ' | tail -1 | sed ' s/%/ / ' )
```

Example questions: process statistics using the ps command

Q For all the processes belonging to the user **donal**, list the names of these processes along with their respective **%CPU** utilisations, in a file called **ps_temp1**, in your home directory.

A solution:

Type:

ps au > ps_temp

Now you have **ps_temp** file that contains the following:

```
USER       PID %CPU %MEM    VSZ  RSS TTY      STAT START   TIME COMMAND
root        3302  0.0  0.0  1700  408 tty1     Ss+  Oct10   0:00 /sbin/mingetty tty1
root       3307  0.0  0.0  2996  408 tty2     Ss+  Oct10   0:00 /sbin/mingetty tty2
root       3308  0.0  0.0  2924  408 tty3     Ss+  Oct10   0:00 /sbin/mingetty tty3
root       3309  0.0  0.0  2640  408 tty4     Ss+  Oct10   0:00 /sbin/mingetty tty4
root       3358  0.0  0.0  1492  408 tty5     Ss+  Oct10   0:00 /sbin/mingetty tty5
root       3407  0.0  0.0  1636  408 tty6     Ss+  Oct10   0:00 /sbin/mingetty tty6
donal       23727  0.0  0.0  6136 1424 pts/1    Ss   09:34   0:00 -bash
donal       23818 49.2  0.0  4220  968 pts/1    R    09:49   0:17 /bin/bash ./busy_loop
donal       23824 48.3  0.0  5516  968 pts/1    R    09:50   0:10 /bin/bash ./busy_loop
donal       23826  0.0  0.0  3824  772 pts/1    R+   09:50   0:00 ps –au
```

To list the processes belonging to **donal** , listing only the **command names** and the **%CPU** utilization, you could use the following command:

awk ' /donal / {print $3,"\t", $11}' ps_temp > ~/ps_temp1
(Note , the "/t" is to insert a TAB)

Where the file **ps_temp1** contains:

```
0.0     -bash
49.2    /bin/bash
48.3    /bin/bash
0.0     ps
```

You could do all the above in a single command line as follows:

ps au > ps_temp ; awk ' /donal / {print $3,"\t", $11}' ps_temp > ~/ps_temp1

or, shorter by using pipes, as follows:

ps au | awk ' /donal / {print $3,"\t", $11}' > ~/ps_temp1

Example problem
Write a bash script file, called **procs_per_user**, that will display the number of processes for each individual user on a system. Note the **ps aux** (or you can use ps –aux) command will provide a full list of processes.

A Solution:

```
#! /bin/bash
#  Display number of process for each user
#  Script name: procs_per_user       DH 3/April/07

# Make list of all users to file: names
ps aux |  awk ' $1 != "USER" {print $1} ' > names

# Make an unique of users in file: uniq_list
sort names | uniq > uniq_list

# Make simple column titles
echo -e "\nProcs  \t Users \n"

# Loop to read each user name and count number of entries
while  read  xuser
do
x=$( grep "$xuser" names |  wc –l )
echo  -e   "$x  \t  $xuser"
done <  uniq_list

# echo new line and exit
echo  -e  "\n"

# remove any temporary files
rm   names   uniq_list

exit 0
```

//

Example result

Procs	Users
1	canna
1	daemon
1	dbus
8	donal
1	gdm
2	higginsm
2	htt
1	jafere
1	nobody

```
117             root
2               rpc
1               rpcuser
1               smmsp
```

Some observations on the above script program:

1) It is good practice (an essential practice) to always remove temporary files before you finish your script program as seen in the above example.

1) In the above example the **uniq_list** file was directed into the body of the **while** loop. Another solution, which might be easier to read, is to use a **for** loop as follows:

```
#! /bin/bash
#  Display number of process for each user .. using a for loop
#  Script name: procs_per_user       DH 26/February/2013

# Make list of all users to file: names
ps aux |  awk ' $1 != "USER" {print $1} ' > names

# Make an unique of users in file: uniq_list
sort names | uniq > uniq_list

# Make simple column titles
echo -e "\nProcs  \t Users \n"

# Loop to read each user name and count number of entries
for xuser in $(cat uniq_list)

do
x=$( grep "$xuser" names | wc -l )
echo  -e   "$x  \t  $xuser"
done

# echo new line and exit
echo  -e  "\n"

# remove any temporary files
rm   names   uniq_list

exit 0
```

Example problem using a signal

Consider the bash script exhibit program as below.

```
# Exhibit program  D.H. 15/March/2012  ver. 1.0.0
#! /bin/bash

# The main code is here
./progB   &     # start program progB in the background

while true      # a simple loop to simulate real activity
do
  echo "I'm looping"
  sleep 1
done
```

Modify this program so that it will include a **signal trap**. The **trap** will do the following:

i) Acts on receipt of a **SIGINT** signal

ii) Contains a function called **trap_function()**

iii) The **function** does the following:

- displays (echoes) a simple message to say what is the **PID** for **progB**
- sends a **HUP** signal to the running **progB** program
- properly exits the script program without **orphaning** progB

A solution

SAMPLE ANSWER TO ABOVE

```
#! /bin/bash

trap  'trap_function'  SIGINT

trap_function()
{
 echo PID for ProgB is "$!"
 kill   -HUP   "$!"
 wait
 exit
}

# The main code is here
```

```
./progB   &     # start program progB in the background
while true      # a simple loop to simulate real activity
do
  echo "I'm looping"
  sleep 1
done
wait
exit
```