82

AIX update

August 2002

In this issue

3 Back-up servers in the DMZ

8 Script to control the printing of PS files based on the number of pages

12 The case statement

22 Performance monitoring using NMON

28 Understanding the uniq command

44 AIX news

AIX Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38342
From USA: 01144 1635 38342
E-mail: trevore@xephon.com

North American office

Xephon
PO Box 350100
Westminster, CO 80035-0100
USA
Telephone: 303 410 9344

Subscriptions and back-issues

A year's subscription to *AIX Update*, comprising twelve monthly issues, costs £180.00 in the UK; $275.00 in the USA and Canada; £186.00 in Europe; £192.00 in Australasia and Japan; and £190.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the November 1999 issue, are available separately to subscribers for £16.00 ($24.00) each including postage.

AIX Update on-line

Code from *AIX Update*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at http://www.xephon.com/aix; you will need to supply a word from the printed issue.

Editors

Trevor Eddolls

Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

Contributions

When Xephon is given copyright, articles published in *AIX Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. To find out more about contributing an article, without any obligation, please download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

Printed in England.

Back-up servers in the DMZ

Taking the mksysb backup images is important. The main problem is when the servers to be backed up (mksysb) are in what is referred to by IBM as a De-Militarized Zone (DMZ), which means it is an area outside the firewall. In this case, the Network File System (NFS) – a client/server application that lets a computer user view and optionally store and update files on a remote computer as though they were on the user's own computer – was not permitted by the firewall. So, we needed another way for the NIM-server to get the mksysb back-up image file from the NIM-client.

For this I wrote two scripts. The first one, mksysb.sh, works on the NIM-client (starts once a week from crontab) and creates the mksysb back-up image file. And the second one, ftp_nim.sh, works on the NIM-server (starts once a week from crontab after the mksysb back-up image file is created) and gets the mksysb file from the NIM-client via FTP. In both scripts, if it completes successfully, it deletes the old files. It then mails the log file to the administrator so he or she can see whether mksysb and FTP have successfully completed. An Outlook mail example is shown at the end of the article.

For the NIM-client, */mksysb* is the directory for mksysb images and log files.

For the NIM-server, */export/images/<server_hostname>* is the directory where the mksysb images are held.

MKSYSB.SH

```
#!/bin/ksh
# Adnan Akbas, 27.01.2002
# The script takes mksysb back-up image file of the server
#  into the directory /mksysb
# variables:
nim_client='hostname'
logfile=/mksysb/mksb_${nim_client}_'date +%H%M_%Y%m%d'.log
mksysb_file=/mksysb/mksb_${nim_client}_'date +%H%M_%Y%m%d'

# Start mksysb
/usr/bin/mksysb '-e' '-i' '-v' '-X' ${mksysb_file} > $logfile 2>&1
```

```
# Checking if the mksysb image file is successfully created
cat $logfile | grep "Backup Completed Successfully" > /dev/null 2>&1
if [ $? -eq Ø ]
then
   print  >> $logfile 2>&1
   echo "MKSYSB image back-up is SUCCESSFULLY created for
${nim_client}" >> $logfile 2>&1

# Changing owner and group to mksysb:staff for ftp.
  /usr/bin/chown mksysb:staff  ${mksysb_file} $logfile >> $logfile 2>&1

# successful,  delete the older version back-up and logfile.
# (older than 3 days)
   print  >> $logfile 2>&1
   echo "Deleting older version image and logfile:" >> $logfile 2>&1
   /usr/bin/find /mksysb -name "mksb_${nim_client}_*" -mtime +3 >>
$logfile 2>&1
   /usr/bin/find /mksysb -name "mksb_${nim_client}_*" -mtime +3 -exec
rm {} \; >> $logfile 2>&1
   print  >> $logfile 2>&1
   exit Ø
else
   echo "MKSYSB image back-up is NOT SUCCESSFULLY created for
${nim_client}" >> $logfile 2>&1
   exit 1
fi
```

FTP_NIM.SH

```
#!/bin/ksh
# Adnan Akbas, 27.Ø1.2ØØ2
# The script gets mksysb image file and its logfile from
#  NIM-clients into /export/images directory via ftp.
# Function that makes ftp for the given NIM-client
function ftp_get {

# Variables
local_dir=/export/images/${nim_client}
state="NOT OK"

# Getting the standard output and error output to logfile
logfile=${local_dir}/mksb_ftp_'date +%H%M_%Y%m%d'.log

# Error messages due to return codes
retstrØ="INFO:  FILES RECEIVED for ${nim_client} !!!"
retstr1="ERROR: Cannot change to ${local_dir} ... FILES NOT RECEIVED
for ${nim_client} !!!"
retstr2="ERROR: Cannot ping  ${nim_client} ... FILES NOT RECEIVED for
${nim_client} !!!"
retstr3="ERROR: Target file does not exist. ... FILES NOT RECEIVED for
```

```
${nim_client} !!!"
retstr4="ERROR: Login failed, check user and password ... FILES NOT
RECEIVED for ${nim_client} !!!"
retstr5="ERROR: Unknown failure ... FILES NOT RECEIVED for
${nim_client} !!!"

# Check if we are in the right directory in NIM-Server
cd ${local_dir} > $logfile 2>&1
if [ $(pwd) != ${local_dir} ]
then
   echo "$retstr1" > $logfile 2>&1
   send_mail
   return
fi

# Check whether the NIM-client is pingable
ping -qc3 ${nim_client} > /dev/null 2>&1
if [ "$?" != "Ø" ]
then
   echo "$retstr2" >> $logfile 2>&1
   send_mail
   return
fi

# Start FTP Job
print  >> $logfile 2>&1
echo "$(date)" >> $logfile 2>&1
print >> $logfile 2>&1
print >> $logfile 2>&1
echo «STARTING FTP ..... « >> $logfile 2>&1
print >> $logfile 2>&1

ftp -v -n ${nim_client} << ! >> $logfile 2>&1
user $user $password
prompt
bin
cd ${target_dir}
ls /mksysb/mksb_*
mget mksb_*
bye
!

print >> $logfile 2>&1
echo "FINISHED: $(date)" >> $logfile 2>&1
print >> $logfile 2>&1

# Checking the output of ftp and determining whether it is successful
# or not. Checking whether the user could log in.
cat $logfile | grep "Login failed"  > /dev/null 2>&1
if [ $? -eq Ø ]
```

```
then
     echo "$retstr4" >> $logfile 2>&1
     send_mail
     return
fi

# Checking whether target directory and file exists. (/mksysb/mksb_*)
cat $logfile | grep "can't find"  > /dev/null 2>&1
if [ $? -eq Ø ]
then
     echo "$retstr3" >> $logfile 2>&1
     send_mail
     return
fi

# Checking whether file is received
[[ 'cat $logfile | grep -c "bytes received in"' -eq 2 ]]
if [ $? -eq Ø ]
then

# If successful, deleting the old version of image and its log file.
   print  >> $logfile 2>&1
   echo "SUCCESSFUL so deleting older version image and logfile:" >>
$logfile 2>&1
   /usr/bin/find ${local_dir} -name "mksb_${nim_client}_*" -mtime +3 >>
$logfile 2>&1
   /usr/bin/find ${local_dir} -name "mksb_${nim_client}_*" -mtime +3 -
exec rm {} \; >> $logfile 2>&1
   print  >> $logfile 2>&1
   print  >> $logfile 2>&1
   echo «$retstrØ» >> $logfile 2>&1

# Checking inside of the received logfile whether the image file is
# successfully created.
   print  >> $logfile 2>&1
   cat ${local_dir}/mksb_${nim_client}_*.log | grep "SUCCESSFULLY" >>
$logfile 2>&1
   state="OK"
   send_mail
   return
fi

# if not returned till here, give unknown failure
echo "$retstr5" >>  $logfile 2>&1
send_mail
return
}

# Function that sends logfile to Outlook mail
function send_mail {
```

```
/usr/bin/mailx -s "${nim_client} - NIM - ftp state: $state" -r NIM-
Server@root adnan.akbas@turkcell.com.tr < $logfile > /dev/null 2>&1
}

# Variables:

# DMZ client hostnames:
dmz_clients="rsc039e0 rsc010e0 rsc011e0 rsc012e0 rsc063e0 rsc024e0"

target_dir=/mksysb
user=mksysb
password=45456

# MAIN ################################################
for nim_client in $dmz_clients
do
  ftp_get
done
#######################################################
```

OUTLOOK MAIL EXAMPLE SENT BY FTP_NIM.SH

```
Mon Apr 15 12:44:43 CEST 2002

STARTING FTP .....

Connected to rsc039e0.
220 rsc039e0 FTP server (Version 4.1 Sun Nov 19 22:15:05 CST 2000)
ready.
331 Password required for mksysb.
230 User mksysb logged in.
Interactive mode off.
200 Type set to I.
250 CWD command successful.
200 PORT command successful.
150 Opening data connection for /mksysb/mksb_*.
/mksysb/mksb_rsc039e0_2000_20020412
/mksysb/mksb_rsc039e0_2000_20020412.log
226 Transfer complete.
200 PORT command successful.
150 Opening data connection for mksb_rsc039e0_2000_20020412 (337971200
bytes).
226 Transfer complete.
337971200 bytes received in 2455 seconds (134.4 Kbytes/s)
local: mksb_rsc039e0_2000_20020412 remote: mksb_rsc039e0_2000_20020412
200 PORT command successful.
150 Opening data connection for mksb_rsc039e0_2000_20020412.log
(1165404 bytes).
226 Transfer complete.
1165404 bytes received in 1.428 seconds (796.9 Kbytes/s)
```

```
local: mksb_rsc039e0_2000_20020412.log remote:
mksb_rsc039e0_2000_20020412.log
221 Goodbye.

FINISHED: Mon Apr 15 13:25:41 CEST 2002

SUCCESSFUL so deleting older version image and logfile:
/export/images/rsc039e0/mksb_rsc039e0_2000_20020405
/export/images/rsc039e0/mksb_rsc039e0_2000_20020405.log

INFO:  FILES RECEIVED for rsc039e0 !!!

MKSYSB image backup is SUCCESSFULLY created for rsc039e0
```

Adnan Akbas
System Administartor
Turkcell (Germany)

Script to control the printing of PS files based on the number of pages

One problem faced by our site regarding printing on AIX was how to control print requests based on the number of pages of ps/prn files. **lpstat** gives the sizes of files in blocks, which is not a good measure for controlling printing because, for example, files containing graphics usually have huge block sizes, but they actually print on a few pages. Below is typical output from the **lpstat** command, which shows a ps file having 13 blocks:

```
Queue   Dev   Status    Job Files       User    PP %   Blks  Cp Rnk
-----   ---   ------    ---------       ----    ----   ----  -- ---
xrxpsq  @xero READY
              SENDING    483 mjcet.ps   root              13   1   1
```

To overcome this problem, we extracted the information about the number of pages by using the **%%Page** parameter (which indicates a new page), from the ps file as shown:

```
grep -w %%Page $print_fname | tail -1 > /tmp/lastpage
no_of_pages='awk '{print $3}' /tmp/lastpage'
```

We have incorporated this feature into our customized print script. Our management-defined policy on printing permits users to print up to 15 pages. The following is the code for our print script, which implements this

policy:

```
#*********************************************************************
#!/bin/ksh
# Customized print script restricting printing of ps/prn files
# over 15 pages
print_stop=Ø                            # reset loop termination flag
print_fname=" "
PRT_STAT="DOWN"
while [[ $print_stop = Ø ]]; do         # loop until done
clear
  cat << PRINT_MENU                     # display menu
                     AIX Printing User Interface

   1  -> Print Duplex PostScript                          $LOGNAME
                                                  'date '+%a %b %d %Y''
   2  -> Exit

PRINT_MENU

  echo 'Enter your choice? '                #prompt
  read print_reply                          #read response

    case $print_reply in
1 )

      clear
      echo " "
      echo " "
      echo "Enter file name to print : "
      read print_fname

if [[ $print_fname != "" ]]
then

   ls -al | grep $print_fname

   if [[ $? != Ø ]]
   then

      clear
      echo " "
      echo " "
      echo " "
      echo You have entered an invalid filename ....
      echo " "
      echo " "
      echo " "
      read enter?'Press ENTER to return to the main menu ....'

   else
```

```
            PCNTR="%\!PS-Adobe"
            TCNTR='head -1 $print_fname | cut -c1-1Ø'
            if [[ ${TCNTR} != ${PCNTR} ]]
            then

            clear
                 echo " "
                 echo " "
                 echo The file is not a PostScript file ....
                 echo " "
                 echo " "
      echo Use this print queue for printing PostScript files only ....
                 echo " "
                 echo " "
      read enter?'Press ENTER to return to the main menu ....'

            else
# This portion of the script checks the number of pages in
# a ps or a prn file
#-----------------------------------------------------------------------
grep -w %%Page $print_fname | tail -1 > /tmp/lastpage
no_of_pages='awk '{print $3}' /tmp/lastpage'
if [ $no_of_pages -gt 15 ]
then
    echo "\n \n The number of pages in $print_fname is $no_of_pages \n"
    echo "In accordance with our Printing Policy, documents larger "
    echo "than 15 pages will not be permitted for printing."
      echo " "
      echo " "
      echo " "
      exit
fi
      PRINTER_STATUS='qchk -Pxrxpsq | tail -1 | awk '{print $3}''
      if [[ ${PRINTER_STATUS} != ${PRT_STAT} ]]
      then
          /usr/bin/qprt -Pxrxpsq -Ban -D$LOGNAME $print_fname

          clear
          echo " "
          echo " "
          echo File queued for printing ....
          echo " "
          echo " "
          read enter?'Press ENTER to return to the main menu ....'

     fi
   fi
fi

else
```

```
        clear
          echo " "
          echo " "
          echo You have entered a blank file name ....
          echo " "
          echo " "
          read enter?'Press ENTER to return to the main menu ....'
fi
;;
    2 )                                  # Exit the Print Program
      print_stop=1
      clear
;;

    * )
              clear
                echo " "
                echo " "
                echo You have entered an invalid choice ....
                echo " "
                echo " "
                echo Enter either 1 or 2
                echo " "
                echo " "
              read enter?'Press ENTER to return to the main menu ....'
;;
    esac
done
```

A sample of the output from the program is shown below. A prn file, Chap3.prn, 1.6MB in size (block size as shown by **lpstat** was 1642), was sent for printing using this script. The print program extracted the number of pages from the ps file as 61. Hence, this file was rejected from being printed.

```
Enter file name to print :
chap3.prn
-rw-r----   1 root     system   1681356 Apr 10 15:55 chap3.prn

 The number of pages in chap3.prn is 61
In accordance with our Printing Policy, documents more than 15 pages
will not be permitted for printing.
```

We hope that this script is useful for sites that intend to control printing of ps/prn files based on the number of pages.

Abul Bashar and Syed Tariq Maghrabi
Systems Analysts
KFUPM (Saudi Arabia)

The case statement

FORMAT OF CASE STATEMENT

The **case** command/statement is an example of a conditional construct which makes it possible to execute a command or sequence of commands in a shell script if a particular condition is true. It provides shell scripts with branching mechanisms and is used to select one command sequence from several alternatives.

You must associate a string pattern with each list of commands to be executed, and also specify the word that is to be compared with the patterns. The shell will compare the word against each of the patterns, and execute the command sequence that is associated with the matching pattern.

The general format of the **case** statement is as follows:

```
case word in
pattern_1)
       command_list_1
       ;;
pattern_2)
       command_list_2
       ;;
.
.
pattern_n)
       command_list_n
       ;;
esac
```

The word **case** introduces the command, and **esac** (case spelt backwards) identifies the end of the command. The words **case** and **esac** are called **keywords**. They are not in themselves commands, but are parts of a command. Generally speaking, for the shell to recognize a keyword, it must be the first word on the line, or it must be the first word after a command terminator, such as a semicolon or an ampersand. The keyword **in** is an exception to this rule.

The word that follows the keyword **case** may be any sequence of characters that the shell can recognize as a single word. You can use the value of a shell variable, such as **$1**, or the output of a command such as **$(date)**.

Each of the command lists, command_list_1 through to command_list_n, may be either a single command, a pipeline, or any sequence of commands and/or pipelines.

Each list of commands except the last must be terminated with two semicolons. In practice, programmers usually terminate each command list, including the last, with the two semicolons.

To execute a **case** command, the shell compares a word with each of the patterns until it finds a matching pattern. The shell then runs the commands in the associated command list. If two patterns happen to match a word, only the commands associated with the first will be run. If none of the patterns match, none of the commands within the **case** structure will be run.

You do not need to format **case** commands exactly as shown. However, it is good practice to indent each command list, since it makes it easier for someone who is reading your shell script to locate each pattern/command list pair.

PATTERN MATCHING

In the **case** command, you form the patterns pattern_1 through to pattern_n in a similar way to forming filename generation patterns, and quite complex constructions can be used to provide unique matches. The pattern matching, however, is purely string against string, and if you want to compare digits and the contents of the word include leading zeros, then you must either get rid of these using **typeset**, or include the leading zeros in the pattern.

There are a few differences, however, between patterns used in a **case** statement and those used in filename generation. For example, you do not need to explicitly match a leading dot or a \, and the pattern ***** will match all strings including those that begin with either a dot or a slash.

You may recall that:

- ***** matches any string of characters, including the empty string. This pattern is usually used as a catch-all for any remaining patterns that do not match the preceding patterns.
- **?** matches any single character.

- **[…]** matches any one of the enclosed characters. If you separate a pair of characters with a hyphen, any character that is lexically between the two is matched.
- **[! . . .]** matches any character except the enclosed characters. For example, **[!0-9]*** matches any string that does not begin with a digit.

If you separate several patterns with pipe characters (|), the associated commands will be executed if the word matches any of the patterns listed. For example, a pattern such as **an*|ca*** will match any word that begins with the letters **an** or any word that begins with the letters **ca**.

EXAMPLE 1 – THE SEARCH SCRIPT

The following example illustrates how the **case** command may be used in a shell script. The script is named **search**, and, depending on what you request, the script will either search for a file, or search for a word within a file:

```
$ vi search

#!/bin/ksh
# Script name: search
# Usage: search
#############################################################
# Version History
# Version      Date      Remarks
# 1.Ø                   Original Version
#############################################################

# determine type of search the user wants
print 'Are you searching'
print 'a: For a file/directory; or'
print 'b: For a string within a file'
print 'Enter your choice: \c'

read answer
case $answer in

a|A) # search for a file
print '\nWhich file are you looking for?: \c'
read file
find / -name "$file" -print 2>/dev/null
;;

b|B) # search for a string within a file
```

```
print '\nWhich file is it in?: \c'
read file
print 'What is the string?: \c'
read string
grep "$string" "$file"
;;

esac
print Good-bye
```

The **case** structure is used to determine what type of search will be performed. You are asked to enter the letter **a** if you wish to search for a file/directory, or the letter **b** if you are looking for a string pattern within a particular file. We use **a|A** and **b|B** as the search patterns since we don't mind if the answer is in lower or upper case.

The **read** command gets your response and puts it in a shell variable named **answer**. Note that the word part of the **case** command is **$answer**, not **answer**.

If you choose to search for a file, the **find** command will be run. In this script, we have specified the directory to start the search at /, so that the entire file system will be searched.

The standard error of **find** has been redirected to */dev/null* since **find** prints a message to standard error each time it encounters a directory that you do not have permission to search; by redirecting standard error, the annoying error messages are no longer displayed.

Notice that we have enclosed **$string** in double quote marks on the **grep** command line. This is to ensure that the value of a variable will be treated as a single argument. Single quotes will not suffice since the **$** will lose its normal metacharacter meaning and **grep** would then search for the characters **$string**, rather than for the characters contained within the **string** variable.

There is a flaw in **search**. It assumes that the user understands from the format of the prompting what response is expected. If you enter anything other than **a** or **b**, your answer will not match either of the patterns. None of the commands within the **case** statement will be run, and you will not be informed that your request was invalid.

This can be remedied by using a default pattern that will be matched when none of the other patterns are matched. The pattern ***** will match any word,

and it must be the last pattern that is listed. Any patterns that follow it will never be matched since the shell executes only the commands associated with the first matching pattern.

To remove this defect, enter the following three lines after the current last semicolon pair:

```
*)
print "search: '$answer' is not a valid choice"
;;
```

You can try out this version of **search**. Note that when you look for a file that produces a large number of matches, for example, if you search for ***font***, the list will scroll off the screen before you have had time to view it. You will see shortly one of the ways in which this can be overcome by redirecting the output of **find** to a temporary file, testing for the existence of entries in the file, and then piping the contents of the file to the **pg** or **more** command; or you can simply pipe the output of **search** to either of these commands.

THE GETOPTS COMMAND

So far we have used relatively simple command line arguments to our scripts. These have usually been straightforward text strings, which have been assigned to **$1**, **$2**, etc. Many Unix commands, however, are much more complex than this and have command line arguments which themselves have further multiple options (arguments).

As a simple example consider the command:

```
chdev -a block_size=1024 -l rmt0
```

Here the **-a** argument has the option **block_size=1024**, and the **-l** argument takes the option **rmt0**. The **-a** argument in particular can take a number of different tape attribute options, whereas the **-l** simply expects the name of a tape drive and doesn't particularly care what this is called. Depending on how an executable has been written, spaces may or may not be permissible between an argument and its options.

To write a script that recognizes where an argument ends and its option starts when there are no spaces between the two requires particularly cumbersome coding. In addition, many commands allow arguments to be combined, **tar -xvf** being an example, and if we wished to do this in a script

with the commands we have learned so far, it would be nigh on impossible, or at the very least exceedingly frustrating, to do so.

Fortunately the shell provides a particularly useful command, **getopts**, for dealing with multiple complex command line arguments, and it is invariably used with a **case** structure. It cannot on its own manage combined arguments, as in the **-xvf** above, and if you write a script where this is required, you will need nested **case** statements to further test the value of the options to any argument.

getopts is by no means perfect as you will discover, but it does allow you to minimize code when processing command line arguments. When used in conjunction with looping and other conditional constructs you should be able to manage the most complex of command lines.

GETOPTS SYNTAX

getopts assumes that command line arguments begin with either a **+** (plus sign) or a **-** (minus sign) followed by a single character. Any characters following the argument that do not begin with either a **+** or a **-**, and it is irrelevant whether they are separated from the argument by spaces or otherwise, are considered to be an option to the argument itself, although you will see later that there are exceptions to this rule under certain error conditions.

There are two arguments to **getopts** itself. The first is an option string that can contain letters and colons, and the second is a variable name which is assigned the value of the argument itself (without the **+** or **-**) every time that **getopts** is executed.

Consider the following example:

```
getopts "v:p:" opt
```

Every time a script containing this line is executed, **getopts** will check whether there are arguments of **-v** or **-p** (we could use the plus sign if we wanted to, but let us assume from now onwards that we are always using the minus sign). The colon following the **v** or **p** tells us that each of these arguments expects an option. If, for example, there was no option to **-v**, then our line would look like:

```
getopts "vp:" opt
```

Unfortunately, **getopts** can check only a single command line argument at a time, so if you wish to check multiple arguments, you must enclose the command in something like a **while** loop (this is covered in a future article), or you must run **getopts** as many times as you expect there to be arguments.

If you enter an invalid argument to your script, for example you use **-x** instead of **-v**, then the **getopts** error message, *A specified flag is not valid for this command*, will be displayed. When this occurs, the variable **opt** is set to the value **?**. However, if your option string starts with a colon, for example, **:v:p:**, then no **getopts** error messages will be displayed. Under these circumstances you must provide your own error message to cover such an eventuality.

If an argument has an option, then this value is stored in the variable **OPTARG**, which you can then use in a following **case** statement. Also, each time **getopts** is invoked it places the index of the next argument to be processed in the variable **OPTIND**. Whenever your script is run, **OPTIND** is initialized to **1**.

EXAMPLE 2 – THE LVMAN SCRIPT

As we said earlier, **getopts** is most often used in conjunction with a **case** statement, so let's now consider a simple example of a script running **getopts** and discuss what happens when you run the script with different arguments. Let us use a script called **lvman**, which is a modified **vgsizes**, and which now contains the following section to check command line arguments:

```
getopts :v:p: opt
case $opt in
v)
      VG=$OPTARG
      f_get_vg_space $VG
      ;;
p)
      LV=$OPTARG
      f_get_pv_space $PV
      ;;
*)
      f_dsp_usage
      exit
      ;;
esac
```

This particular version assumes that we run our script either as **lvman -v vg_name** or as **lvman -p pv_name**. If we were to use **+v** instead of **-v** then we would have to explicitly use **+v)** for our pattern to be matched.

If we run our script with **lvman -v rootvg**, say, **OPTARG** will be set to **rootvg**, the variable **VG** will then be set to the same value, and then the function **f_get_vg_space rootvg** will be called. There is a similar logic when the script is run with the **-p** option. You could, if you wished, call the functions using, for example, **f_get_vg_space $OPTARG**.

Should we enter an invalid argument, **opt** would take the value **?**, and our catch-all test at the end would call the function to display a usage message. We could have used **\?)** as our pattern to be matched, but as you will see from the table below, this produces different messages depending on how you run the script.

```
      getopts>      getopts v:p: opt      getopts :v:p: opt
      case stanza>      *)
   f_dsp_usage      \?)
   f_dsp_usage      *)
   f_dsp_usage      \?)
   f_dsp_usage
      lvman        or
lvman xx      getopts error
Usage message      getopts error
Usage message      Usage message      Usage message
 lvman -v  or lvman -p Usage message  Usage message   Usage message  No
messages
 lvman -v -p      No messages      No messages      No messages      No
messages
 lvman -x getopts error getopts error Usage message Usage message
```

The commands in the first column are assumed to have been run without options to their arguments, or apart from **lvman xx**, without any arguments. The second and third columns show the output when either a ***** or **\?** are used as the matching patterns when **getopts** is called without the leading colon in the options string. In the third and fourth columns we are using the leading colon when **getopts** is called.

Generally speaking, the **getopts** error messages produced in addition to a usage message are unnecessary, and so the leading colon is preferable in the option string to remove these messages. You can also see that the ***** matching pattern will produce the greatest number of usage messages and this is the preferred pattern to use.

If you were to run the command **lvman -v -p**, then no matter which pattern you use, no error or usage messages will be generated. When this happens **OPTARG** is set to the value **-p** and your code should include an additional test to spot this error; we shall see in a future article how this can be done.

In addition to the **getopts** section, a number of modifications have been made to the original **vgsizes** script to give it a more modular construction using functions, and the current version of **lvman** will look as follows:

```
#!/bin/ksh
# Script name: lvman
# Usage: lvman {[-v VGname] | [-p PVname]}
################################################################
# Version History
# Version      Date      Remarks
# 1.Ø                   Original Version
################################################################

#-----------------------------------------------
# Function: f_dsp_usage
# Displays usage messages
#-----------------------------------------------

f_dsp_usage()
{
      print "Usage: $(basename $Ø) {[-v VGname] | [-p PVname]}"
      print "Where:"
      print "\t-v VGname specifies a single volume group"
      print "\t-p PVname specifies a single physical volume"
      print "Note: Use either the -v OR -p option"
}

#-----------------------------------------------
# Function: f_get_vg_space
# Arguments: $1 - volume group name
# Gets the total and free space of the volume group
#-----------------------------------------------

f_get_vg_space()
{
      VG=$1
      #
      # Get total space and free space
      #
      TOTAL=$(lsvg $VG | grep "TOTAL PPs" | cut -f2 -d "(" |
            tr ' ' '\t' | cut -f1)
      FREE=$(lsvg $VG | grep "FREE PPs" | cut -f2 -d "(" |
```

```
            tr ' ' '\t' | cut -f1)
      eval ${VG}_LVNUM=$(lsvg -l $VG |tail +3 |wc -l |tr -d " ")
      #
      # Print output
      #
      printf "%-20s %-15s %-15s\n" "Volume Group" \
"Total Size" "Free Space"
      printf "%-20s %-15s %-15s \n" $VG "$TOTAL MB" \
"$FREE MB"
      eval print Number of LVs in $VG = '$'${VG}_LVNUM
}

#---------------------------------------------
# Function: f_get_pv_space
# Arguments: $1 - physical volume name
# Gets the total and free space on a physical volume
#---------------------------------------------

f_get_pv_space()
{
      PV=$1
      #
      # Get total space and free space
      #
      TOTAL=$(lspv $PV | grep "TOTAL PPs" | cut -f2 -d "(" |
            tr ' ' '\t' | cut -f1)
      FREE=$(lspv $PV | grep "FREE PPs" | cut -f2 -d "(" |
            tr ' ' '\t' | cut -f1)
      eval ${PV}_LVNUM=$(lspv -l $PV |tail +3 |wc -l |tr -d " ")
      #
      # Print output
      #
      printf "%-20s %-15s %-15s\n" "Physical Volume" \
"Total Size" "Free Space"
      printf "%-20s %-15s %-15s \n" $PV "$TOTAL MB" \
"$FREE MB"
      eval print Number of LVs on $PV = '$'${PV}_LVNUM
}

#########################################################
# Main section
#########################################################

getopts :v:p: opt
case $opt in
v)
      VG=$OPTARG
      f_get_vg_space $VG
      ;;
p)
      PV=$OPTARG
```

```
        f_get_pv_space $PV
        ;;
*)
        f_dsp_usage
        exit
        ;;
esac
```

You will note that the **f_dsp_usage** function contains the construction **$(basename $0)**. If the full pathname of the script was */usr/local/bin/lvman* and we executed the script using this full pathname, then the **basename** command would extract only the characters after the last /; under most circumstances we would not want the full pathname to be displayed in the usage statement.

The script also no longer uses the source file, **vgs**, and the **FREE** and **TOTAL** values are now extracted directly from the output of the **lsvg** and **lspv** commands. This slows down the execution of the script slightly, but instead ensures that we will always have accurate data for our volume groups and physical volumes and we will not have to recreate the **vgs** file prior to running the script.

Tonto Kowalski
Guru (UAE)

Performance monitoring using NMON

NMON is system performance monitoring tool written and maintained by Nigel Griffith (nag@uk.ibm.com) . The separate binary executable of the tool is available for the following versions of AIX:

- nmon_aix415 – AIX 4.1.5
- nmon_aix420 – AIX 4.2.0
- nmon_aix432 – AIX 4.3.2
- nmon – AIX 4.3.3
- nmon – AIX 5.1 32 bit kernel
- nmon64 – AIX 5.1 64 bit kernel.

The current version of the tool is 6f and it is obtainable by direct download from http://www-1.ibm.com/servers/esdd/articles/analyze_aix/agree_down.html.

You should note that IBM does not provide official support for the tool. However, the tool is updated about every six months or when a major release of AIX is announced.

INSTALLATION PROCESS AND REQUIREMENTS

The tool is a stand-alone binary file. After downloading the binary file from the abovementioned URL, you execute the command:

```
tar xvf nmon.tar
```

and execute the binary file suitable for your OS system version. The default interval to update the statistics displayed by the tool is two seconds, causing very little additional load on the system.

In order to enable usage of the tool by regular users you should enable read access to the */dev/kmen* device by executing the command:

```
chmod ugo+r /dev/kmem
```

In order to enable the monitoring of disk statistics, the following command should be executed by a root user:

```
chdev -l sys0 -a iostat=true
```

PROGRAM FEATURES

The program's user interface is based on textual and graphical screens displayed inside a terminal or terminal emulation window. For the best performance you should use a **dtterm** window with the largest possible size – this will allow large amounts of information, highlighted with colour, to be displayed.

Nmon enables a system administrator to monitor and display the following performance-related data:

- Global CPU utilization as well as utilization of separate CPUs.
- Memory usage.

- Kernel and run queue statistics.
- Disk I/O rates, transfers, and read/write ratios.
- Disk activity map.
- VM parameters.
- Amount of free space in filesystems.
- Disk adapters statistics.
- Network I/O rates, transfers, and read/write ratios.
- Paging space utilization and paging statistics.
- CPU and AIX OS details.
- Top processes.
- Usage and utilization of IBM HTTP Web cache.
- Settings and usage of asynchronous I/O processes.
- Consolidated multipathing activity for disks using EMC, Autopath, or SDD.
- Verbose mode – monitoring of vital system statistics against arbitrarily defined limits in order to detect potential performance bottlenecks.

INTERACTIVE OPERATION OF THE PROGRAM

Selection of statistics to be displayed can be done by specifying single letter common line arguments or by typing the same letter inside the program's display window. Subsequent typing of the same letter toggles the display of the data on and off.

For brief help information type **nmon -?**, for full details type **nmon –h**.

SAMPLE NMON OUTPUT

```
Kernel Internal Statistics    (all per second)
RunQueue=   12.Ø swapIn =      2.5 iget  =       18.5  namei = 4Ø46.6
pswitch = 34Ø9.5 syscall= 514547.7 rawch =       78.Ø  canch =    Ø.Ø
fork    =   34.5 read   =   4Ø8Ø.6 dirblk=     1224.8  outch =  549.2
exec    =   4Ø.5 write  =   3491.4 readch = 4542945.9     R+W=    4.3MB
```

```
msg     =    0.0 sem    =     16.0 writech= 2337067.5

Adapter I/O read   write       xfers  Adapter Type
2D-08        0.0   500.8 kB/s   50.0  Wide/Fast-20 SCSI I/O Controller
27-08-01     0.0  2000.8 kB/s  300.9  FC SCSI I/O Controller Protocol D
3A-08-01     0.0  2000.0 kB/s  250.9  FC SCSI I/O Controller Protocol D
TOTALS       0.0  4501.6 kB/s  601.8  TOTAL=  4501.6

Network I/O
I/F Name Recv  Trans kB/s  packin  packout  insize  outsize
     en0 532.1 281.8       2117.0   2038.0   257.4  141.616.0
     en1   2.4  11.4         35.0     62.5    71.6  186.5
     lo0   0.4   0.4          2.5      2.5   152.6  152.6

Verbose Mode
 Code    Resource            Stats   Now        Warn    Danger
     OK -> CPU               %busy  71.5%       >80%    >90%
Warning -> Paging Space      %free  88.5%       VM<RAM  <20%
Warning -> Page Faults       faults 2198.8      >12/s   >120/s
     OK -> Top Disk   hdisk4 %busy   1.0%       >40%    >60%
```

OPERATION OF PROGRAM IN DATA CAPTURING MODE

The program is able to record collected performance data in CSV (Comma Separated Values) format, which can be loaded into a spreadsheet program for further analysis and graphing. Use the following **nmon** flags to specify this mode of operation:

- *-f* – spreadsheet output format (with the following defaults: -s300 –c288). The output filename is <hostname>_YYYMMDD_HHMM.nmon.
- *-F <filename>* – specify output file name.
- *-l <number>* – specify the number of disks per sheet (default – 150, maximum – 250).
- *-r <name>* – specify the name of the spreadsheet file (default – hostname).
- *-t* – include top processes in the output.
- *-s <seconds>* – specify interval between measurements.
- *-c <number>* – specify number of snapshots.
- *-x* – specify capacity planning mode (-fdt –s900 –c96 means take

measurements every 15 minutes for 1 day).

Note that the generated file has to be sorted before loading to the spreadsheet. The sample AIX command is:

```
sort -A hostname_20020501_1300.nmon -o hostname.csv
```

NMON_ANALYZER USAGE

NMON_Analyser has been designed by Stephen Atkins (steve_atkins@uk.ibm.com) to analyse data captured by **nmon** in the CSV files. A separate spreadsheet is available for Lotus 1-2-3 and Microsoft Excel. A single file containing both, as well as a detailed user's guide, can be downloaded from http://www-1.ibm.com/servers/esdd/articles/nmon_analyser/agree_down.html.

After loading the supplied analyser template into the spreadsheet program, the user should specify a number of options set as values for the following spreadsheet cells found in Analyzer and NLS sheets.

After completion of this step, the user can press a button that will start a dialog allowing him to load the captured data file and begin the analysis. The next step is to save the generated spreadsheet that contains a number of useful graphs and tables.

The following is a list of reports produced by the tool:

- Calculation of weighted averages for hot-spot analysis.
- Distribution of CPU utilization by processor over the collection interval – useful in identifying single-threaded processes.
- Additional sections for vpaths showing device busy, read transfer size, and write transfer size by time of day.
- Total system data rate by time of day, adjusted to exclude double-counting of EMC hdiskpower devices – useful in identifying I/O subsystem and SAN bottlenecks.
- Separate sheets for EMC hdiskpower devices.
- Analysis of memory utilization to show the split between computational and non-computational pages.

- Total data rates for each network adapter by time of day.
- Summary data for the TOP section showing average CPU and memory utilization for most active processes.

SAMPLE NMON_ANALYZE OUTPUT

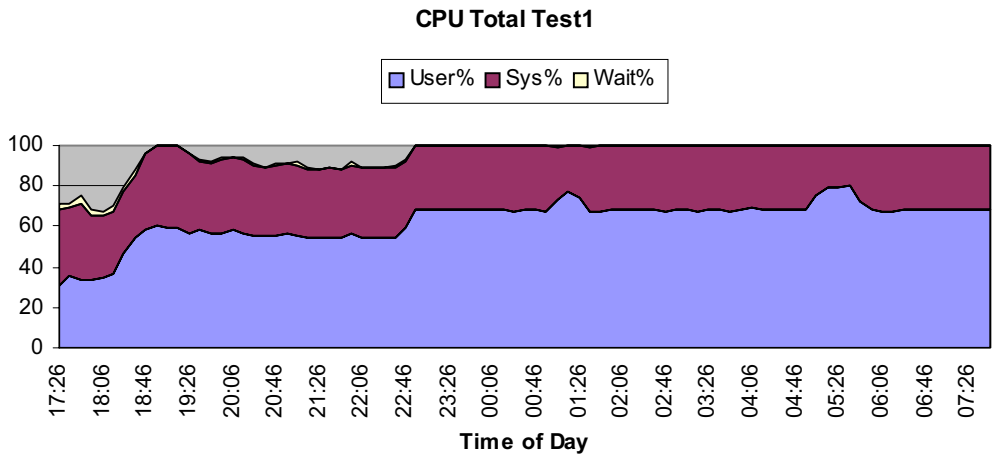

Figure 1: CPU utilization

Figure 1 shows CPU utilization distribution during the collection period.

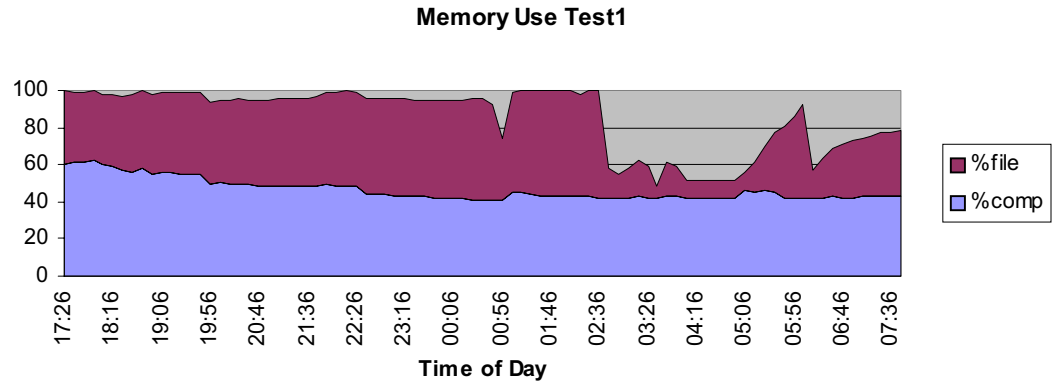

Figure 2: Memory usage

Figure 2 shows a distribution of usage of memory between file and computational (executable program) pages.

Alex Polak
System Engineer
APS (Israel)

Understanding the uniq command

UNIQ COMMAND BASICS

The **uniq** command is a very useful AIX tool that displays or deletes repeated lines in a file. Although making files smaller by deleting duplicate entries is the command's primary purpose, the value of the **uniq** command becomes apparent when used in your diagnostic work.

The **uniq** command can help you isolate lines or sections from logs, lists, tables, or other text data sources. It can count duplicated lines and can show lines that have no duplicates for your diagnostic purposes.

Before the **uniq** command can be run, it is important to sort the input file using the **sort** command. The **uniq** command processes adjacent duplicate lines but cannot locate duplicates found if unique lines have been encountered between the duplicates.

The basic syntax of the **uniq** command is shown below:

```
uniq flags infile outfile
```

where:

- *flags* is an optional flag or flags used to enhance the **uniq** operation.
- *infile* is the file or files on which the **uniq** operation is to perform.
- *outfile* is an optional file to which the output is to be written. If no *outfile* is specified, **uniq** will write to the display.

ELIMINATING DUPLICATES USING THE UNIQ COMMAND

Use the **uniq** command to display a file with any duplicated lines reduced to a single occurrence only.

Suppose you had a file with 300 lines in it. Each line contains the userid of a person who has responded to a live forum, in the chronological order they responded, with the text of their response removed.

For example, let's say that the first several lines of forum_log contain:

```
busybody
frank32
anderk
frank32
bainesw
anderk
busybody
dovera
dovera
frank32
```

You want to display a list of the userids of all the respondents without regard to the number of times their entries appear.

First you must sort the list so the **uniq** command can see all duplications as adjacent entries. If you were to enter:

```
sort forum_log > forum_sort
```

the file forum_sort would contain:

```
anderk
anderk
bainesw
busybody
busybody
dovera
dovera
frank32
frank32
frank32
```

If you were then to enter:

```
uniq forum_sort forum_uniq
```

the file forum_uniq would contain:

```
anderk
bainesw
busybody
dovera
frank32
```

The **uniq** command ran through the sorted list and removed all but one of any entries found adjacent to a duplicate. This would show you the userids of the participants in the forum. Note that you must specify the > redirection symbol on the sort command but not on the **uniq** command.

FLAGS FOR THE UNIQ COMMAND

The following flags extend the usefulness of the **uniq** command:

- *-u* – displays only unique lines. This flag will disregard any lines if there are duplicated lines elsewhere in the file.
- *-d* – displays only non-unique (duplicated) lines. This flag will disregard (not display) any lines that have no duplicates in the file. One line will be displayed for each entry found to have one or more duplications.
- *-c* – counts the number of duplicates found. In front of each displayed line is a numeric count of how many times that line appears, even if it is only once (a unique line).
- *-f Fields* – fields to skip while checking for duplicates. The **uniq** command will begin inspecting lines following the blank delimited field specified.
- *-s Chars* – characters to skip while checking for duplicates. The **uniq** command will begin inspecting lines following the count of characters specified.

The following sections provide further clarification and examples of the usage of flags for the **uniq** command.

SHOWING NO DUPLICATES (-U FLAG)

Use the **-u** flag with the **uniq** command to display only the unique lines without regard to duplicated lines. This tells the **uniq** command to ignore any lines if there are duplicates in the file.

Suppose you had a file containing 1000 entries, and you know that most are duplicated; you know that there are just a few lines without duplicates, and those are the ones you want to see. Use the **-u** flag for that purpose.

Or suppose you expect all lines to have one or more duplicates. You can use the **-u** flag to verify your theory. In this case, any output produced using the **-u** flag would indicate an error condition. In your diagnostic work, you can find that using the **uniq** command to validate your theories is invaluable.

You can even use the **-u** flag to compare two files, as illustrated in the following two examples, one showing lines containing no duplicates and the other verifying that all lines have a duplicate.

Example 1 (show lines containing no duplicates):

You have two lists, one with 500 lines, and one with 501 lines. The latter list is the same as the first with one extra line. One is a log from a working machine and the other is a log from a failing machine. You want to know the contents of that extra line, but do not know where in the file it resides.

If you were to concatenate the two lists together, then sort the results and pass them to the **uniq** command with the **-u** flag, the result would be the single line that was unique between the two files.

Simplifying the example, suppose the working_log contains:

```
"Process has begun"
"Step One has begun"
"Step One has completed"
"Step Two has begun"
"Step Two has completed"
"Process has ended"
```

and the failing_log contains:

```
"Process has begun"
"Step One has begun"
"Step One has completed"
"Step Two has begun"
"Recovery process has been initiated"
"Step Two has completed"
"Process has ended"
```

If you were to enter:

```
cat working_log failing_log | sort | uniq -u
```

the result would be:

```
"Recovery process has been initiated"
```

To help clarify the example, let's look at the sorted intermediate file:

```
"Process has begun"
"Process has begun"
"Process has ended"
"Process has ended"
```

```
"Recovery process has been initiated"
"Step One has begun"
"Step One has begun"
"Step One has completed"
"Step One has completed"
"Step Two has begun"
"Step Two has begun"
"Step Two has completed"
"Step Two has completed"
```

Note that “Recovery process has been initiated” is the only line not containing a duplicate in the list, and, therefore, that is the line that would be displayed as the output.

Example 2 (verify that all lines have a duplicate):

You have two lists you believe to be identical in content, although the order of the entries is not necessarily the same between the two files. One is an alphabetic list of messages from a messages manual. The other is a test log supposedly displaying all the messages, not in alphabetic order, but in order of occurrence. You want to know if they contain the same data.

If you were to use the **uniq** command with the **-u** flag on the combined, sorted contents, the result would yield any lines unique to either file, or the return of your cursor if the files are the same.

Simplifying the example, suppose ordered_list contains:

```
"AB1200 File name not found"
"AB1600 File name too long"
"EF0020 Data type incorrect"
"EF0080 Data not found"
"IJ500  Field cannot be blank"
"IJ600  Field length exceeded"
"OP5555 Numeric data expected"
"OP6666 Numeric information missing"
```

and test_log contains:

```
"OP6666 Numeric information missing"
"EF0080 Data not found"
"IJ500  Field cannot be blank"
"EF0020 Data type incorrect"
"AB1200 File name not found"
"IJ600  Field length exceeded"
"OP5555 Numeric data expected"
"AB1600 File name too long"
```

If you were to enter:

```
cat ordered_list test_log | sort | uniq -u
```

the result would be the return of your cursor. Although the contents are in a different order, they contain the same lines, thus there are no unique lines between the two.

To help clarify the example, let's look at the sorted intermediate file:

```
"AB1200 File name not found"
"AB1200 File name not found"
"AB1600 File name too long"
"AB1600 File name too long"
"EF0020 Data type incorrect"
"EF0020 Data type incorrect"
"EF0080 Data not found"
"EF0080 Data not found"
"IJ500  Field cannot be blank"
"IJ500  Field cannot be blank"
"IJ600  Field length exceeded"
"IJ600  Field length exceeded"
"OP5555 Numeric data expected"
"OP5555 Numeric data expected"
"OP6666 Numeric information missing"
"OP6666 Numeric information missing"
```

Note that each line contains at least one duplicate, therefore there are no lines unique to the sorted intermediate file. This would confirm your theory that the two lists are identical in content.

SHOWING DUPLICATES ONLY (-D FLAG)

Use the **-d** flag with the **uniq** command to display only duplicated lines without regard to unique lines (lines with no duplicates). This tells the **uniq** command to ignore any lines that are not repeated somewhere in the file.

Suppose you had a file that contained 800 entries, most of them unique. You know there are only a dozen or so lines that are duplicated, and it is those lines you want to inspect. The **-d** flag will help you.

Or suppose you expect that all the lines are unique, and you want to verify the fact. Using the **-d** flag will help confirm your theory. In this case, any output produced using the **-d** flag would indicate an error condition.

You can even use the **-d** flag to compare two files, as illustrated in the following two examples, one showing duplicates and the other verifying that no lines have a duplicate.

Example 1 (show duplicated lines):

You have two lists, one with 120 lines, and one with 35 lines. The lists have only one line in common, but you don't know which line. The first is a list of graduates of a certain school. The latter is a list of department members of a work group. You need to find the work group member who graduated from that school.

If you were to concatenate the two lists together, then sort the results and pass them to the **uniq** command with the **-d** flag, the result would be the single line that was shared by the two files.

Simplifying the example, suppose graduates_list contains:

```
Anderson, Kenneth
Baines, William
Charles, Beatrice
Dover, Ann
Edwards, Elizabeth
Franklin, Randolph
Grover, Martin
Hillibrand, Edward
```

and dept_members contains:

```
Allistair, Benjamin
Bilford, Martin
Claires, Milicent
Dover, Ann
Edmonson, James
Fitzgerald, Cynthia
```

If you were to enter:

```
cat graduates_list dept_members | sort | uniq -d
```

the result would be:

```
Dover, Ann
```

To help clarify the example, let's look at the sorted intermediate file:

```
Allistair, Benjamin
Anderson, Kenneth
```

```
Baines, William
Bilford, Martin
Charles, Beatrice
Claires, Milicent
Dover, Ann
Dover, Ann
Edmonson, James
Edwards, Elizabeth
Fitzgerald, Cynthia
Franklin, Randolph
Grover, Martin
Hillibrand, Edward
```

Note that the only duplicated line is “Dover, Ann”, therefore it is that line which would represent the only line in common between both files.

Example 2 (verify that no lines have a duplicate):

You have two lists you believe share no common lines, but you need to make sure. One is a list of words in a standard dictionary and the other is a list of words in a supplementary dictionary. There is no need for duplicate entries between the two.

If you were to use the **uniq** command with the **-d** flag on the combined, sorted contents, the result would yield any lines shared between the files.

Simplifying the example, suppose standard_dict contains:

```
application
batch
compile
data
edit
format
```

and suppl_dict contains:

```
anodization
base_process
catalyst
distilling
energize
fluid_method
```

If you were to enter:

```
cat standard_dict suppl_dict | sort | uniq -d
```

the result would be the return of your cursor. There are no common lines

to display.

To help clarify the example, let's look at the sorted intermediate file:

```
anodization
application
base_process
batch
catalyst
compile
data
distilling
edit
energize
fluid_method
format
```

Note that there are no duplicated lines, so a request to show only duplicated lines would yield no results. This would confirm your theory that the two lists share no entries.

SHOWING COUNTS OF DUPLICATED LINES (-C FLAG)

Use the **-c** flag with the **uniq** command to display a count of the duplicated lines. This could be useful if you needed to know the numbers of times certain error messages appeared in a log, for example.

Example 1:

You have a log of 500 incoming internal telephone calls and you need to determine from which extensions you were dialled most frequently.

Simplifying the example, suppose incoming_phonelog contains:

```
x5555  Dover, Ann
x5555  Dover, Ann
x5002  Edmonson, James
x5120  Bilford, Martin
x5979  Claires, Milicent
x5111  Fitzgerald, Cynthia
x5619  Allistair, Benjamin
x5120  Bilford, Martin
x5555  Dover, Ann
x5120  Bilford, Martin
x5555  Dover, Ann
x5002  Edmonson, James
x5111  Fitzgerald, Cynthia
```

If you were to enter:

```
cat incoming_phonelog | sort | uniq -c
```

the result would be:

```
2 x5002  Edmonson, James
2 x5111  Fitzgerald, Cynthia
3 x5120  Bilford, Martin
4 x5555  Dover, Ann
1 x5619  Allistair, Benjamin
1 x5979  Claires, Milicent
```

To help clarify the example, let's look at the sorted intermediate file:

```
x5002  Edmonson, James
x5002  Edmonson, James
x5111  Fitzgerald, Cynthia
x5111  Fitzgerald, Cynthia
x5120  Bilford, Martin
x5120  Bilford, Martin
x5120  Bilford, Martin
x5555  Dover, Ann
x5555  Dover, Ann
x5555  Dover, Ann
x5555  Dover, Ann
x5619  Allistair, Benjamin
x5979  Claires, Milicent
```

Note that the quantities of each extension in the sorted file match the quantities found in the output of the **uniq** example. The count would help you determine the most and least frequent callers.

SKIPPING FIELDS (-F FLAG)

Use the **-f** flag with the **uniq** command to ignore the specified number of fields when checking for duplicates.

Suppose you had a log with hundreds of entries. Each entry starts with a common string identifying an error condition and the name of a file. The error text following the leading identifier can be repeated throughout the log. You want to delete the duplicated error text descriptions without regard to the name of the file against which the error was reported.

For example, let's say that the first several lines of error_log contain:

```
File c:\uniqexer\tst in error:   "File name not found"
```

```
File c:\uniqexer\testfile.txt in error:   "Data type incorrect"
File c:\uniqexer\tst in error:   "Field cannot be blank"
File c:\uniqexer\newfil in error:   "File name not found"
File c:\uniqexer\newfil in error:   "Field cannot be blank"
File c:\uniqexer\tst in error:   "Field cannot be blank"
File c:\uniqexer\testfile.txt in error:   "Numeric information missing"
File c:\uniqexer\testfile.txt in error:   "File name not found"
File c:\uniqexer\newfil in error:   "Field cannot be blank"
```

The first step would be to sort the error log after the last colon (:), which is in the fourth, blank delimited field.

Sorting error_log after field 4 into error_sort with the command:

```
sort +4 error_log > error_sort
```

would yield:

```
File c:\uniqexer\testfile.txt in error:   "Data type incorrect"
File c:\uniqexer\newfil in error:   "Field cannot be blank"
File c:\uniqexer\newfil in error:   "Field cannot be blank"
File c:\uniqexer\tst in error:   "Field cannot be blank"
File c:\uniqexer\tst in error:   "Field cannot be blank"
File c:\uniqexer\newfil in error:   "File name not found"
File c:\uniqexer\testfile.txt in error:   "File name not found"
File c:\uniqexer\tst in error:   "File name not found"
File c:\uniqexer\testfile.txt in error:   "Numeric information missing"
```

Note that now all data beginning with the first double quote (") is alphabetically sorted.

Entering:

```
uniq -f 4 error_sort error_uniq
```

would yield:

```
File c:\uniqexer\testfile.txt in error:   "Data type incorrect"
File c:\uniqexer\newfil in error:   "Field cannot be blank"
File c:\uniqexer\newfil in error:   "File name not found"
File c:\uniqexer\testfile.txt in error:   "Numeric information missing"
```

This would show you one occurrence of each error message without regard to the number of times it occurred nor the file for which it occurred. Since the **uniq** command skipped the first four fields of each entry, the file names of the results would be meaningless – the critical data are the error messages only.

SKIPPING CHARACTERS (-S FLAG)

Use the **-s** flag with the **uniq** command to ignore the specified number of characters when checking for duplicates. Sometimes the data you want the **uniq** command to ignore does not land on a field boundary. In those cases, you may find that specifying a character boundary can give you the desired results.

Suppose you had a log that contained many temperature readings, each of which is preceded by a time and date stamp. For example, let's say that the first several lines of temp_log contain:

```
11:05:00 15/06/2002___20C
12:05:00 15/06/2002___22C
15:05:00 15/06/2002___25C
11:05:00 16/06/2002___20C
13:05:00 16/06/2002___22C
14:05:00 16/06/2002___24C
14:05:00 17/06/2002___25C
15:05:00 17/06/2002___26C
```

Note that each entry has a series of underscores between the date and the temperature reading. This would prevent you from specifying a field count to the **uniq** command. You want to display the unique temperature readings without regard to the time or date the reading was taken.

Sorting the log after 22 characters into temp_sort using the command:

```
sort -k 1.22 temp_log > temp_sort
```

would yield:

```
11:05:00 15/06/2002___20C
11:05:00 16/06/2002___20C
12:05:00 15/06/2002___22C
13:05:00 16/06/2002___22C
14:05:00 16/06/2002___24C
14:05:00 17/06/2002___25C
15:05:00 15/06/2002___25C
15:05:00 17/06/2002___26C
```

Note that now all the data following the last underscore is alphabetically sorted. Entering:

```
uniq -s 22 temp_sort temp_uniq
```

would yield:

```
11:05:00 15/06/2002___20C
12:05:00 15/06/2002___22C
14:05:00 16/06/2002___24C
14:05:00 17/06/2002___25C
15:05:00 17/06/2002___26C
```

This would tell you there were five individual temperature readings logged at various times and dates in the file. Since the **uniq** command skipped the first 22 characters of each entry, the time and date stamps of the results would be meaningless – the critical data is the temperatures only.

SOME EXERCISES

Here are some exercises to test your knowledge of the **uniq** command.

Exercise 1 – eliminating duplicates

Step 1 – set up an exercise file by entering the following text into a file called exer1.fil:

```
dogs
dogs
cats
squirrels
cats
bears
bears
dogs
```

Step 2 – enter:

```
cat exer1.fil | sort | uniq
```

and note that the result will be:

```
bears
cats
dogs
squirrels
```

You have used the **uniq** command to eliminate duplicates of the text in the exercise file.

Exercise 2 – showing no duplicates (-u flag)

Step 1 – take a log file with several hundred entries and create a test file for

this exercise. For example, enter:

```
cat verylarge.log > exer2.fil
```

Step 2 – to ensure each line in the exercise file has at least one duplicate, enter:

```
cat verylarge.log >> exer2.fil
```

which will append the entire contents of the log back into the exercise file.

Step 3 – enter:

```
cat exer2.fil | sort | uniq -u
```

and note that the result is the return of your cursor. This is because there were no lines in the exercise file that did not have a duplicate somewhere in the file.

Step 4 – now edit the exercise file and insert a few lines that are unique to the entries in the file, and unique to each other, such as your name or the city in which you work.

Step 5 – enter:

```
cat exer2.fil | sort | uniq -u
```

and note that the results are only those lines you added.

Note: if you had chosen to CHANGE some lines rather than to add unique lines, the results would be the new lines you had changed, plus the corresponding unchanged lines because now they are unique!

Exercise 3 – showing duplicates only (-d flag)

Step 1 – take a log file with several hundred entries that you know in advance are unique, such as a log with timestamps over a range of dates, and create a test file for this exercise. For example, enter:

```
cat timestamp.log > exer3.fil
```

Step 2 – enter:

```
cat exer3.fil | sort | uniq -d
```

and note that the result is the return of your cursor. This is because there were no lines in the exercise file that were duplicated anywhere in the file.

Step 3 – now edit the exercise file and duplicate a few of the lines.

Step 4 – enter:

```
cat exer3.fil | sort | uniq -d
```

and note that the results are only a single occurrence each of those lines you duplicated.

Exercise 4 – showing counts of duplicated lines (-c flag)

Step 1 – create an exercise file as follows. Start with three small files, each with several lines of unique data. Concatenate the first file into the exercise file once, the second file into the exercise file twice, and the third file into the exercise file three times. For example:

```
cat fileone > exer4.fil
cat filetwo >> exer4.fil
cat filetwo >> exer4.fil
cat filethree >> exer4.fil
cat filethree >> exer4.fil
cat filethree >> exer4.fil
```

The result of your set-up will be an exercise file with a few unique lines and several duplicate lines with two or more occurrences each.

Step 2 – enter:

```
cat exer4.fil | sort | uniq -c
```

and note that the result is a table with a linecount of occurrences of each line found in the file.

Step 3 – enter:

```
cat exer4.fil | sort | uniq -c | sort -r
```

and note that the result is similar to Step 2, only now the results are sorted in order of occurrences, greatest to least. Appending **sort -r** to your command can be useful if your intent is to display a table with the most meaningful data at the top or bottom.

Exercise 5 – skipping fields (-f flag)

Step 1 – set up an exercise file by entering the following text into a file called exer5.fil:

```
Line One text: "Using the uniq command helps your diagnostic work."
Line Two text: "You can compare two files with the uniq command."
Line Three text: "It is simple to use the uniq command."
Line Four text: "Using the uniq command helps your diagnostic work."
Line Five text: "It is simple to use the uniq command."
Line Six text: "You can compare two files with the uniq command."
```

Step 2 – enter:

```
cat exer5.fil | sort +3 | uniq -f 3
```

and note that the result will be:

```
Line Five text: "It is simple to use the uniq command."
Line Four text: "Using the uniq command helps your diagnostic work."
Line Six text: "You can compare two files with the uniq command."
```

You have used the **-f** flag to eliminate duplicates of the text following the first three fields of each line in the file. Remember that the data preceding the colon (:) is meaningless in the output.

Exercise 6 – skipping characters (-s flag)

Step 1 – set up an exercise file by entering the following text into a file called exer6.fil. Ensure no leading blanks precede any entries:

```
Line 1 Data=001002003
Line 2 Data=abcabcabc
Line 3 Data=xyzxyzxyz
Line 4 Data=abcabcabc
Line 5 Data=999999999
Line 6 Data=001002003
```

Step 2 – enter:

```
cat exer6.fil | sort -k 1.12 | uniq -s 12
```

and note that the result will be:

```
Line 1 Data=001002003
Line 5 Data=999999999
Line 2 Data=abcabcabc
Line 3 Data=xyzxyzxyz
```

You have used the **-s** flag to eliminate duplicates of the text following the first 12 characters of each line in the file. Remember that the data preceding the equal sign (=) is meaningless in the output.

David Chakmakian
Programmer (USA)

AIX news

IBM has introduced a host of new high-performance features in its AIX operating system that are designed to supercharge compute-intensive applications. The company also announced that it will become the first major Unix vendor to offer a 'productized' version of the Globus Toolkit, the industry's *de facto* standard open source Grid management software.

The new features are said to improve the speed of bandwidth-intensive workloads, such as BI applications that search massive corporate data warehouses, as well as HPC applications, such as simulation.

The new features include large data transfer, enabling bigger chunks of information to be accessed more efficiently in the computer's memory. AIX offers support for the traditional 4KB page size and for the new 16MB 'large page' size and localization, where processors running a particular workload have optimized access to system memory components, further increasing performance.

IBM has also announced plans to introduce an AIX toolbox for Grid applications, based on the open source protocols from Globus, the recognized leaders in Grid management software. The toolbox, which IBM is offering free of charge, is middleware that allows users to share supercomputing power, data, and applications as easily as information is shared over the Web.

For further information contact your local IBM representative.
URL: http://www-1.ibm.com/servers/aix/news/supercharge.html.

* * *

IBM has launched Infoprint Manager for AIX Version 4 Release 1, which is designed to help manage and monitor printers, provide intelligent document routing and scheduling, and help balance printer workloads.

Enhancements in Infoprint Manager for AIX include enhanced calibration support for colour printers and the ability to hold a processing job and move it to a different printer. It now provides count of actual number of pages printed, not just a projected count, and supplies actual counts of pages drawn from the input trays.

It now reports the number of copies requested and provides job completion date and time, and there's improved job completion notification for SAP 3.1h.

The security function provides greater granularity for non-DCE environments, having the ability to define the kinds of messages each user should get, and supporting colour e-mail output.

There's the ability to specify a separate overlay for each document in a job ticket in Infoprint Submit, the ability to print SAP output with AFP resources, such as overlays, images, and barcodes, support for PCL 6, and wireless notification of job status.

For further information contact your local IBM representative.
URL: http://www.printers.ibm.com/R5PSC.NSF/Web/ipmnewaix41.

* * *