93

AIX update

July 2003

In this issue

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.

Controlling shark mirror locations

In case of a disaster, most companies want to keep their data in different locations and cities. One of the best ways is to mirror logical volumes on shark storage to those in different locations. For example, our company, which has shark storage in both Cologne and Düsseldorf, prefers to mirror their central logical volumes in Cologne to shark storage in Düsseldorf. The problem is, because many system administrators have access to the servers, sometimes when they create new logical volumes, they do not create the mirrors on the shark storage at the other location. That's why I have written a script that checks whether all logical volumes on shark systems are correctly mirrored and lists the ones which are not. Additionally, it checks the state of the logical volumes and warns if the mirrors are not synchronized. This script can be executed on all servers without having to be changed.

To make life easier (easy to update), I keep only one list on a particular server, which shows the locations of the shark storage, and, when I execute the script on a server, first of all it makes an ftp connection as an anonymous user to the particular server to get the list.

List: shark_location.txt:

```
SHARK 1 13173 CGN
SHARK 2 13755 CGN
SHARK 3 14826 DUS
SHARK 4 12324 CGN
SHARK 5 12096 CGN
SHARK 6 18482 DUS
SHARK 7 16352 CGN
```

Looking at the first line we have the following information:

- SHARK 1 – the label number of the shark storage.
- 13173 – the shark_id.
- CGN – location Cologne.

The most important command used here is **lsvpcfg**, which shows the assignment of the hdisk to the vpath (vpath is a logical grouping of hdisks).

An example of **lsvpcfg** output is:

```
vpathØ (Avail pv TØ4vg) 5ØØ12324 = hdisk6 (Avail ) hdisk19 …
vpath1 (Avail pv TØ4vg) 5Ø112324 = hdisk7 (Avail ) hdisk2Ø …
vpath2 (Avail pv TØ4vg) 4ØØ12324 = hdisk8 (Avail ) hdisk21 …
vpath3 (Avail pv TØ4vg) 4Ø112324 = hdisk9 (Avail ) hdisk22 …
vpath4 (Avail pv TØ4vg) 7ØØ12324 = hdisk1Ø (Avail ) hdisk23 …
vpath5 (Avail pv TØ4vg) 7Ø112324 = hdisk11 (Avail ) hdisk24 …
vpath6 (Avail pv TØ4vg) 6ØØ12324 = hdisk12 (Avail ) hdisk25 …
vpath7 (Avail pv TØ4vg) 6Ø112324 = hdisk13 (Avail ) hdisk29 …
vpath11 (Avail pv TØ4vg) 1ØE18482 = hdisk17 (Avail ) hdisk3Ø …
vpath12 (Avail pv TØ4vg) 2Ø418482 = hdisk18 (Avail ) hdisk31 …
vpath8 (Avail pv TØ4vg) 11312324 = hdisk14 (Avail ) hdisk16 …
vpath9 (Avail pv TØ4vg) 11412324 = hdisk15 (Avail ) hdisk26 …
```

Considering the first line we see:

- vpath0 – showing which vpath number the shark storage uses.
- 5001234 – the last five digits forming the shark_id.

With this information, first of all I know which shark storage (shark_ids) is connected to the server, and, with the list I transferred, I can identify where they are located. After knowing which shark storage uses which vpaths I can check my mirrored logical volumes one by one and their vpaths to see whether they are located in different shark systems in different locations.

Vpath0...9 are for SHARK1 and vpath11,12 are for SHARK6 in this case. After checking the vpaths of logical volumes, we expect the logical volumes on SHARK1 to have mirrors on SHARK6. This is checked by writing the vpaths of shark storage in CGN to a file and vpaths of shark storage in DUS to another file. Then after getting vpaths of the mirrored logical volume, we must see that one of the original vpaths must be in a CGN file while the other mirror vpath should be in the DUS file.

A CGN file might be (orig_file in script):

```
vpathØ
vpath1
```

```
vpath2
vpath3
vpath4
vpath5
vpath6
vpath7
vpath8
vpath9
```

And a DUS file (mirr_file in script) might be:

```
vpath11
vpath12
```

Then checking the vpaths of the logical volume:

```
# lslv -m o_saplo1_lv | grep vpath | awk '{print $3, $5}' | sort | uniq

vpath0 vpath11
vpath1 vpath11
vpath2 vpath12
vpath3 vpath12
vpath4 vpath12
vpath5 vpath12
vpath6 vpath11
vpath7 vpath11
vpath8 vpath12
vpath9 vpath11
```

(The first column shows the original, the second column the mirror.)

As shown above, the mirrors are correctly created. The originals are between vpath0 and vpath9 on SHARK1 Cologne, and the mirror copies are vpath11 and vpath12 on SHARK6 Düsseldorf.

Another useful feature of the script is the logical volume state check.

For example:

```
# lsvg –l DERvg
DERvg:
LV NAME         TYPE       LPs   PPs   PVs  LV STATE      MOUNT POINT
logDERvg        jfslog     4     8     4    open/syncd    N/A
oraclelv        jfs        1     2     2    open/syncd    /oracle
o_817_64_lv     jfs        44    88    8    open/syncd    /DER/817_64
o_saparc_lv     jfs        128   256   20   open/syncd    /DER/saparch
o_p04_lv        jfs        41Ú×  88    20   open/syncd    /DER
```

```
o_sdtØ2_lv     jfs        1968  3936  26   open/syncd    /DER/sapdata2
o_sdtØ3_lv     jfs        1948  3896  26   open/syncd    /DER/sapdata3
o_sdtØ4_lv     jfs        2148  4296  26   open/syncd    /DER/sapdata4
o_sdtØ5_lv     jfs        1979  3958  26   open/syncd    /DER/sapdata5
o_sdtØ6_lv     jfs        1811  3622  26   open/syncd    /DER/sapdata6
o_saplo1_lv    jfs        2Ø    4Ø    26   open/syncd    /DER/saplog1
o_sappag_lv    jfs        16    32    26   open/syncd    /DER/sappage
o_sapreo_lv    jfs        32    64    26   open/syncd    /DER/sapreorg
sarchlv        jfs        68    136   26   open/syncd    /SER/archiv
o_sdtØ1_lv     jfs        2Ø66  4132  26   open/syncd    /DER/sapdata1
```

Opened/stale indicates that the logical volume is open but contains partitions that are not current.

Opened/syncd indicates that the logical volume is open and synchronized.

Closed indicates that the logical volume has not been opened. The script warns also when the logical volumes are not synchronized.

SHARK_CHK_MIRROR.SH

```
#!/bin/ksh
# Adnan Akbas , Turkcell , Ø9.Ø9.2ØØ2
#
# This script checks whether the logical volumes on the shark storage
# are mirrored on the shark storage at a different location and
# lists the ones that are not correctly mirrored.
#######################################
# This function receives the shark list from a server in order to
# get updated list containing the shark storages and their locations
function get_shark_list {
# change directory to determine where to ftp
cd $local_dir > $logfile 2>&1
if [[ $(pwd) != $local_dir ]]
then
   echo $retstr1
   exit 1
fi
# Check if service is pingable
ping -c 2 $target 1 > /dev/null 2>&1
if [[ $? != Ø ]]
then
   echo $retstr2
   exit 2
fi
# Start FTP
```

```
ftp -v -n $target << ! >> $logfile
user $user $passwd
prompt
bin
pwd
cd $target_dir
pwd
ls
hash
get $shark_file
bye
!
# Check if the file exists and not empty
if [ -s ${local_dir}/${shark_file} ]
then
   print
   echo $outstr1 >> $logfile
   print
else
   echo $retstr3
   exit 3
fi
echo "---------------------------------" >> $logfile
print >> $logfile
}
#######################################
# This function writes mirror & original vpaths in separate files.
function get_vpaths {
# Removing files created by the previous execution
rm $orig_file > /dev/null 2>&1
rm $mirr_file > /dev/null 2>&1
# determine which sharks ids are connected to the server
# and identify their locations. Then writing mirror and
# original vpaths to two different files.
lsvpcfg | awk -F= '{print $1}' | awk '{print $5}' | cut -c4-8 | sort |
uniq | grep . |
while read shark_id
do
 loc='grep $shark_id ${local_dir}/${shark_file} | awk '{print $4}''
 if [[ $loc = "CGN" ]]
 then
   lsvpcfg | grep $shark_id | awk '{print $1}' >> $orig_file
 elif [[ $loc = "DUS" ]]
 then
   lsvpcfg | grep $shark_id | awk '{print $1}' >> $mirr_file
 else
   echo $retstr4
   exit 4
 fi
done
```

```
}
#####################################
# This function checks whether lv mirrors are at a different location
function lv_check {
# finding which volume groups are on shark storage
lspv | grep vpath | awk '{print $3}' | sort | uniq |
while read vg
do
# finding all logical volumes having mirrors
  lsvg -l $vg | grep jfs | awk '{print $1, $6}' |
  while read lv syn junk
  do
     numcp='lslv $lv | grep COPIES: | awk '{print $2}''
     if [[ $numcp = 2 ]] then
# finding which vpaths the lvs and their mirrors have
#  and comparing their locations.
       lslv -m $lv | grep vpath | awk '{print $3, $5}' | sort | uniq |
       while read vp1 vp2 junk ; do
     hit_orig=$(('grep -xc $vp1 $orig_file'+'grep -xc $vp2 $orig_file'))
     hit_mirr=$(('grep -xc $vp1 $mirr_file'+'grep -xc $vp2 $mirr_file'))
         if [[ $hit_orig != 1 || $hit_mirr != 1 ]]
         then
           echo "$vg - $lv - LOGICAL VOLUME MIRROR IN THE SAME
LOCATION!" | tee -a $logfile
           flag=false
           break
         fi
       done
     fi
     lv_state='echo $syn | awk -F/ '{print $2}''
# Checks the lv state and warns if the mirror is not synchronized
     if [[ $lv_state != "syncd" ]]
     then
       echo "$vg - $lv - LOGICAL VOLUME STATE: $syn" | tee -a $logfile
       flag=false
     fi
  done
done
}
#####################################
# This function checks the flag and gives out an exit code
function chk {
if $flag
then
  print | tee -a $logfile
  echo $outstr2 | tee -a $logfile
  exit Ø
else
  print | tee -a $logfile
  echo $retstr5 | tee -a $logfile
```

```
  exit 5
fi
}
# variables ##########################
local_dir=/tmp
target=1Ø.63.11.12
logfile=${local_dir}/shark.log
orig_file=${local_dir}/orig.file
mirr_file=${local_dir}/mirr.file
target_dir=pub
shark_file=shark_location.txt
user=anonymous
passwd=${HOSTNAME}@turkcell.com.tr
flag=true
# error/info codes ###################
retstr1="ERROR: Cannot change to $local_dir! FILE NOT RCEIVED !!!"
retstr2="ERROR: Cannot ping $target! FILE NOT RECEIVED !!!"
retstr3="ERROR: $shark_file NOT SUCCESSFULLY RECEIVED !!!"
retstr4="ERROR: $shark_file NOT CORRECT !!!"
retstr5="ERROR: The logical volumes above must be corrected !!!"
outstr1="INFO: $shark_file SUCCESSFULLY RECEIVED ..."
outstr2="INFO: Logical volumes are correctly mirrored on this machine
..."
# main ###############################
get_shark_list
get_vpaths
lv_check
chk
#####################################
```

Adnan Akbas
System Administrator
Turkcell (Germany)

More illustrated usage of various shell commands and shell features

This article provides more examples of shell commands and features, which I hope others will find useful.

FTP

Perform an ftp operation in batch mode:

```
TP_NODE=mastst
SOURCE_FILE="/export/home/zamana/sh/a.dat"
REMOTE_FILE="/export/home/zamana/temp/a.dat"
#
FTP_COMM_FILE="/tmp/ftp.dat"
FTP_ERROR_FILE="/tmp/ftp.err"
#
USER="zamana"
PWD="power01"
#
cat  <<! > ${FTP_COMM_FILE}
open  ${FTP_NODE}
user ${USER} ${PWD}
put ${SOURCE_FILE} ${REMOTE_FILE}
bye
!
#
ftp -n <${FTP_COMM_FILE} > ${FTP_ERROR_FILE} 2>&1
#
if [ -s  ${FTP_ERROR_FILE} ]
then
     echo "FTP  failed "
else
     echo "FTP succeeded"
fi
#
rm ${FTP_COMM_FILE}
rm ${FTP_ERROR_FILE}
#
```

Notes:

- Create a script as described above and execute it to perform the ftp.
- The **-n** option with ftp tells ftp not to try an automatic logon.

For ftp to perform an ftp operation in batch mode with automatic login, there must exist a *.netrc* file in the home directory of the ftp initiator on the server from which ftp is being initiated.

The contents of the *.netrc* file are:

```
machine  mastst  login  zamana  password  power01

#TP_NODE=mastst
#SOURCE_FILE="/export/home/zamana/sh/a.dat"
#REMOTE_FILE="/export/home/zamana/temp/a.dat"
#
```

```
FTP_COMM_FILE="/tmp/ftp.dat"
FTP_ERROR_FILE="/tmp/ftp.err"
#
#USER="zamana"
#PWD="powerØ1"
#
cat  <<! > ${FTP_COMM_FILE}
put ${SOURCE_FILE} ${REMOTE_FILE}
bye
!
#
ftp mastst <${FTP_COMM_FILE} > ${FTP_ERROR_FILE} 2>&1
#
if [ -s  ${FTP_ERROR_FILE} ]
then
     echo "FTP  failed "
else
     echo "FTP succeeded"
fi
#
rm ${FTP_COMM_FILE}
rm ${FTP_ERROR_FILE}
#
```

Notes:

- Create a script as described above and execute it to perform the ftp.
- **ftp mastst** tells the ftp to initiate an automatic login using the contents of *.netrc* file.
- The contents of *.netrc* file are as follows:

```
machine   <  remote_host_name>   login  < remote_userid>
password  < remote_userid_password>
```

WAIT

Run a script in the background from within a script and wait for it to finish:

```
ERROR_FILE=/tmp/process_invoice.err
#
echo " Enter Invoice Period\c"
read   INVOICE_PERIOD
#
# launch process_invoice executable in the background
```

```
# and wait for it to complete
#
process_invoice   &
#
#  store the process id
#
PID="$!"
#  wait for the process to finish
#
wait  ($PID)
#
#  check the error file
#
if  [  -s   ${ERROR_FILE}  ]
then
         echo "Process failed"
else
         echo "Process succeeded"
fi
```

Note: without the **wait** command, the process will fall through to checking the error file, which will not produce the correct check.

TAR

Make a tar file containing all the files in the current directory:

```
CUR_DIR=/u1/product/oracle/8.1.7
PARENT_DIR=/u1/product/oracle
#
TAR_FILE="ora817.tar"
#
cd   ${PARENT_DIR}

tar  -cf    ${TAR_FILE}     ./8.1.7/*
```

Note: the tar file contains relative path name ./8.1.7. When this tar file is opened, a directory called 8.1.7 will be created under the directory from which the **tar** command was initiated.

Open a tar file with a relative path name:

```
PRODUCT_DIR=/u1/product/oracle
#
#  we want to open the tar file under directory ${PRODUCT_DIR}.
#
#  copy the tar file from, let's say, /tmp, into this directory
#
```

```
cp     /tmp/${TAR_FILE}     ${PRODUCT_DIR}
#
tar  -xf    ${TAR_FILE}
```

TEE

Send output to a file as well as to a terminal:

```
LOG_FILE=/tmp/f1.log
touch   ${LOG_FILE}
echo  "Initializing …."   |  tee  -a    ${LOG_FILE}
```

TRUSS

Generate a trace file for running an executable:

```
SOURCE_FILE=a.c
EXE_FILE=a.out
TRACE_FILE=truss.out
```

Listing for $SOURCE_FILE:

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <curses.h>
main ()
{
pid_t  pid ;
pid = fork () ;
if  ( pid > Ø  )
{
  printf("Parent\n");
}
else
{
  printf("Child\n");
}
}
```

Generation of a trace file:

```
truss  -o   truss.out   -f   ${EXE_FILE}
```

Listing for ${TRACE_FILE}:

```
21883:  execve("a.out", ØxFFBEF96C, ØxFFBEF974)  argc = 1
21883:  resolvepath("/usr/lib/ld.so.1", "/usr/lib/ld.so.1", 1Ø23) = 16
21883:  open("/var/ld/ld.config", O_RDONLY)            Err#2 ENOENT
```

```
21883:  open("/usr/lib/libc.so.1", O_RDONLY)            = 3
21883:  fstat(3, 0xFFBEEFB4)                            = 0
21883:  mmap(0x00000000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0xFF3A0
000
21883:  mmap(0x00000000, 778240, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0)
= 0xFF2
80000
21883:  mmap(0xFF334000, 32736, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_
FIXED, 3, 671744) = 0xFF334000
21883:  open("/dev/zero", O_RDONLY)                     = 4
21883:  mmap(0xFF33C000, 6216, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_F
IXED, 4, 0) = 0xFF33C000
21883:  munmap(0xFF326000, 57344)                       = 0
21883:  memcntl(0xFF280000, 131820, MC_ADVISE, 0x0003, 0, 0) = 0
21883:  close(3)                                        = 0
21883:  open("/usr/lib/libdl.so.1", O_RDONLY)           = 3
21883:  fstat(3, 0xFFBEEFB4)                            = 0
21883:  mmap(0xFF3A0000, 8192, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED, 3, 0)
 = 0xFF3A0000
21883:  close(3)                                        = 0
21883:  open("/usr/platform/SUNW,Ultra-Enterprise-10000/lib/
libc_psr.so.1", O_RD
ONLY) = 3
21883:  fstat(3, 0xFFBEEE0C)                            = 0
21883:  mmap(0x00000000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0xFF390
000
21883:  mmap(0x00000000, 16384, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0)
= 0xFF38
0000
21883:  mmap(0x00000000, 8192, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE, 4, 0
) = 0xFF370000
21883:  close(3)                                        = 0
21883:  close(4)                                        = 0
21883:  munmap(0xFF390000, 8192)                        = 0
21883:  fork()                                          = 21884
21884:  fork()          (returning as child ...)        = 21883
21883:  ioctl(1, TCGETA, 0xFFBEE9E4)                    = 0
21884:  ioctl(1, TCGETA, 0xFFBEE9E4)                    = 0
21883:  write(1, " P a r e n t\n", 7)                   = 7
21883:  llseek(0, 0, SEEK_CUR)                          = 88660
21884:  write(1, " C h i l d\n", 6)                    = 6
21883:  _exit(1)
21884:  llseek(0, 0, SEEK_CUR)                          = 88660
21884:  _exit(1)
```

Note: the options used with the **truss** command are as follows:

- **-o** – output file name.
- **-f** – follow child process as well.

CAT

Number the records in a file:

```
INPUT_FILE=/u1/zamana/f1_not_numbered.dat
OUTPUT_FILE=/u1/zamana/f1_numbered.dat

cat  -n   ${INPUT_FILE}  >  ${OUTPUT_FILE}
```

FILE

Find out about a file type:

```
QUERY_FILE=/tmp/f1

echo   "xxxxxx"  >  ${QUERY_FILE}

file   ${QUERY_FILE}
```

Output:

```
f1:    ascii   text
```

Or:

```
SOURCE_FILE=/tmp/a.c
QUERY_FILE=/tmp/a.out
cc  -o   ${QUERY_FILE}  ${SOURCE_FILE}
file   ${QUERY_FILE}
```

Output:

```
a.out:     executable
```

SIZE

Find out the number of bytes allocated for different segments in an executable:

```
SOURCE_FILE=/tmp/a.c
QUERY_FILE=/tmp/a.out
cc  -o   ${QUERY_FILE}  ${SOURCE_FILE}
```

```
size    ${QUERY_FILE}
```

Output:

```
1222 +  344  + Ø  =  1566
```

Notes:

- The first number is the number of bytes allocated for text.
- The second number is the number of bytes allocated for data.
- The third number is the number of bytes allocated for uninitialized data.
- The fourth number is the total number of bytes allocated to the file, which may or may not be equal to the size shown by the **ls –l** command because the space allocation for a file is made using a number of blocks that is a constant until changed.

SCRIPT

Capture everything from the screen:

```
script
```

Notes:

- The command will initialize a file called typescript in the current directory and write everything from the screen into this file.
- Press **ctrl-d** to terminate the process.

MKFIFO

Make a pipe file to transfer a file's contents to be compressed:

```
PIPE_FILE=/tmp/pipe.dat
COMPRESSED_FILE=/tmp/f1.Z
mkfifo  ${PIPE_FILE}
compress  < ${PIPE_FILE}  >  ${COMPRESSED_FILE}
```

Notes:

- The **compress** command will read from file, ${PIPE_FILE} and compress the content into ${COMPRESSED_FILE}.
- Another process must exist that will write to ${PIPE_FILE}.
- The ideal use of this is a situation where a large intermediate file needs to be generated before being compressed (eg an Oracle export file).

DIFF AND ED

Generate a diff file suitable for use with the **ed** command in order to re-create a changed file:

```
ORI_FILE=/tmp/ori_file.dat
FILE_VERSION1=/tmp/file2.dat
DIFF_FILE=/tmp/changes.dat
```

Description of $ORI_FILE:

```
This article is an illustrated usage of various commands.
Each command is explained in the context of its usage.
```

Description of $FILE_VERSION1:

```
This article is an illustrated usage of various commands
and features.
Each command is explained in the context of its usage.
The article is written by Arif Zaman.
```

Generation of diff file:

```
diff -e ${ORI_FILE} ${FILE_VERSION1} > ${DIFF_FILE}
```

Notes:

- The file ${VERSION_CHANGES} contains commands that are suitable for use with the **ed** command to generate the Version1 file from the original file.
- The contents of ${DIFF_FILE} are as follows:

```
2a
The article is written by Arif Zaman.
.
1c
```

```
This article is an illustrated usage of various commands
and features.
.
```

- To generate Version1 of the file from the original file, do the following:
  - Add the command **1,$p** to the end of file ${DIFF_FILE}.
  - Run the following command:

```
ed - ${ORI_FILE} < ${DIFF_FILE} > ${FILE_VERSION1}
```

- Notice changes are applied in reverse order, with the changes later in the file appearing first. This is essential whenever you're making changes based on line numbers; otherwise, changes made earlier in the file may change the numbering, rendering the later parts of the script ineffective.

Apply multiple diff files in order to re-create the required version of an original file:

```
ORI_FILE=/tmp/ori_file.dat
FILE_VERSION1=/tmp/file1.dat
FILE_VERSION2=/tmp/file2.dat
VERSION1_CHANGES=/tmp/ver1_changes.dat
VERSION2_CHANGES=/tmp/ver2_changes.dat
```

Description of $FILE_VERSION2:

```
This article is an illustrated usage of various commands
and features.
Each command is explained in the context of at least one of its usages.
The article is written by Arif Zaman.
```

Generating Version1 changes:

```
diff -e ${ORI_FILE} ${FILE_VERSION1} > ${VERSION1_CHANGES}
```

Generating Version2 changes:

```
diff -e ${FILE_VERSION1} ${FILE_VERSION2} > ${VERSION2_CHANGES}
```

Generation of Version2 file:

```
( cat ${VERSION1_CHANGES} ${VERSION2_CHANGES} ; echo '1,$p' ) | ed -
${ORI_FILE} \
Ø    ${FILE_VERSION2}
```

OD

Dump contents of a file that might contain certain control characters:

```
FILE_NAME=/u1/zamana/name.dat

od -c ${FILE_NAME}
```

Note: any control characters will be displayed with either familiar notation (eg line feed with \n) or the ASCII equivalent number (eg 004 or 033).

Arif Zaman
ETL Developer (UK)

Regular expressions and pattern matching

So far in this series of articles, our major exposure to pattern matching and regular expressions has been via the **case** and **expr** commands. You may already be familiar with the relatively simple usage of filename generation characters, but in this article we are going to devote ourselves to the more complex regular expressions that a number of commands use to match text, usually deploying the patterns against the contents of a particular file, or against their standard input.

Filename generation can also be an integral part of shell programming. For example, your script may want to generate a list of filenames on which you want to perform a number of operations. The way in which files are matched to generate your list, using **ls** say, can be quite different from matching text using the regular expressions understood by other commands, and where necessary we will show how the metacharacters differ in their usage.

There are a number of commands involved in the extraction and manipulation of text using regular expressions, consisting mostly of metacharacters, but, just to keep you on your toes, the syntax

used in expressions varies from command to command; some metacharacters used by one command may not be available in others, and some perform different functions depending on the command in which they are used. To add to the confusion there are commands such as the **case** statement, which appear to have a mix of regular expressions and filename generation-type matches.

You are undoubtedly familiar with **grep**, but you may not be familiar with its stable mate, **egrep**, which uses an extended set of metacharacters for pattern matching, or **fgrep**, which is equivalent to **grep -F** and treats each pattern as a string, rather than a regular expression. The family of **grep**s are most commonly used to match text in files. **sed** and **awk** are two very useful commands that rely heavily on the use of regular expressions, particularly **sed**. And let's not forget everybody's favourite, **vi**, which is an extension of **ed**, the line editor, and which makes extensive use of pattern matching when you use its search and replace operators.

Regular expressions allow you to write simple or complex patterns, but because of the differences in interpretation by commands, and possibly also your own misconceived level of understanding of regular expressions, not to mention making simple errors, you can quite often achieve matches that are not what you expect. If your script does something irreversible with its matches, then be absolutely certain you know what you are doing. The rule is, 'test the pattern to see what it matches before using it in earnest'. You have been warned!

To understand pattern matching and regular expressions you must become familiar with the functions that each metacharacter performs, and our discussions will, by necessity, involve not only the complex expressions, with which you may be unfamiliar, but also simple pattern matching expressions which you probably know already, although we will not dwell too long on them since we don't want to bore you to death! So, if some of the material is already known to you, then my sincere apologies.

A regular expression is a sequence of characters which may contain any of the standard alphanumeric characters, in addition to a number of metacharacters having their own particular matching abilities. We may also want to match metacharacters themselves, and these will have to be escaped to ensure that they are treated as ordinary characters, not metacharacters. You should not think of regular expressions as matching words, but matching individual characters, which may themselves constitute words, but not necessarily so.

To illustrate this, let us consider an example of the **tr** command which tends to be used with a simple form of pattern matching. If we used **tr** with the following syntax:

```
tr 'cat' 'dog' . .
```

it would certainly change every instance of cat to dog, which is what we may have wanted to do, but the translation is not of words, but of characters, so that every **c** will be changed to a **d**, every **a** to an **o**, and every **t** to a **g**, so that the word **that** will be changed to the completely incomprehensible **ghog**. Merely matching patterns is often the easy bit, but changing the match to something different, and making sure that you have also not changed anything else, is the hard part.

Metacharacter functions in pattern matching

Metacharacters used in regular expressions and pattern matching perform three basic functions, although there may be variations from these:

- Matching literal characters themselves.
- Grouping characters into strings for matching, or placing them into classes to match any one of a number of characters.
- Acting as operators to match zero, one, or more, of a single character, or group of characters; in filename generation, some of these operators also match literal characters.

In the following sections we will consider the different metacharacters and the function each performs, whether it be for use in regular expressions, or in filename generation. When we talk about escaping metacharacters to convert them into ordinary characters, we mean preceding them, in most cases, with a backslash, ****. Just about every command seems to interpret the backslash as an escape character, whereas, usually due to syntax reasons, we often cannot escape the metacharacters by enclosing them in single quotes, say.

MATCHING WITH . (DOT)

The dot metacharacter is used to match literal characters. In regular expressions it is effectively a wildcard that matches any single character, except a newline. For example, **a.c** matches **abc**, **a9c**, **a:c**, etc.

To match the dot itself during regular expressions it must be escaped, but when used in filename generation this is not necessary since it only matches the character **.** (dot).

MATCHING WITH [...]

This construction is used to match a class of literal characters and matches any one, or more, of the characters enclosed between the brackets. For example, **[abc]** matches any pattern containing **a**, or **b**, or **c**, or **ab**, or **ac**, etc. If you wanted to match any combination of **abc** followed by the numbers **1** or **2**, or both, then you could use the pattern **[abc][12]**. The pattern **[abc12]** would achieve the same thing, but could also contain matches that consisted of the numbers only, or of the letters only, which is not what we want.

If you wish to match a bracket itself in your regular expression, using **grep** for example, you can neither quote it, nor escape it with a single backslash; any attempts to do this will result in an imbalance error message and you can only match brackets by escaping them with a double backslash, quoting the single backslash followed by the bracket, or by enclosing them in

brackets themselves. For example:

```
grep [[]
grep '\['
grep "\["
grep \\[
```

The double backslash method means that the first backslash escapes the second, and then forces the shell to pass a single backslash followed by a bracket to the **grep** command. The single backslash now tells **grep** to treat the following character as a literal character, rather than as a metacharacter. Similarly, the quoted backslash and bracket are passed as a whole to **grep** to interpret them in the same way.

There are certain peculiarities when you use the brackets method to match a single bracket, and the actual location within the surrounding brackets can be quite important, depending on what you are trying to achieve. For example, suppose you have a file containing the lines:

```
line with [
line with ]
line with a
```

Then:

grep []] matches line with *]*.

grep []a] matches line with *]* and line with *a*.

grep [a]] matches nothing.

grep [[] matches line with *[*.

grep [[a] matches line with *[* and line with *a*.

grep [a[] matches line with *[* and line with *a*.

The rule seems to be ‘the bracket you want to match must be the first character within the surrounding brackets’. This applies, of course, only when you want to match a single bracket. Either of the above two methods for matching brackets can result in some strange-looking regular expressions. For example, if you have a variable **var** containing the string **abc[def]ghi**, then to use **expr**

to extract the brackets and their contents from the variable, we must use the syntax:

```
expr $var : ".*\(\\[.*\\]\).*"
```

or:

```
expr $var : ".*\([[].*[]]\).*"
```

Instead of using **grep** for matching brackets and other similar metacharacters, however, it can often be better to use **fgrep** to avoid using escape characters since the bracket is interpreted by **fgrep** as a simple string character, not a metacharacter. As you will see, there are also a small number of metacharacters available with **egrep**, and which can also be used with **grep** provided they are escaped. For readability it is perhaps better to use **egrep** in these cases.

Matching ranges

A hyphen (**-**) is used to match a range of characters according to the current collating sequence. For example, **[a-z]** will match any lower-case alphabetic character. Other useful ranges are **[A-Z]** for all upper case characters, **[0-9]** for all digits, and any combinations such as **[a-zA-Z]**, or **[a-z0-9]**.

You should be aware that the pattern **[a-Z]** will not match all upper and lower case characters since **A** to **Z** come before **a** to **z** in the ASCII collating sequence, and so the pattern will match only **a**, **-**, and **Z**. The similar pattern, **[A-z]**, will match all the upper and lower case characters, but also a number of other characters that fall within this collating sequence.

If you want to match the hyphen itself, this must be either the first or last character in the expression, such as **[-+*/]**, for example, or you can use a backslash to escape it.

Excluding characters

Within our braces we can not only specify which characters we want to match, we can also specify the characters we want to

exclude from matches. We do this in regular expressions by using the caret, **^**, which must be the first character after the leading bracket in order for the shell to interpret it as an exclusion character. If the caret appears anywhere else in the brackets, or has been escaped, it is treated as an ordinary character.

When excluding characters during filename generation, instead of the caret we use the exclamation mark, !, as our exclusion character. The same comments regarding positioning are applicable.

It is often easier to use an exclusion expression than to specify pattern lists of all the characters you want to match. For example, if we wanted to include all non-alphabetic characters, it is easier to use the pattern [^0-9a-zA-Z] or [!0-9a-zA-Z] for filename generation (strange filenames!) than it is to explicitly state all the characters you want to match.

MATCHING WITH &

The ampersand (&) is a metacharacter that is used in a small number of commands to represent the remembered characters matched by the regular expression. It is used only in commands such as **sed** and **vi**, which have substitute operations to allow character strings to be modified and replaced. In such circumstances, the **&** can be used as a shorthand notation to represent the whole of the matched pattern, which you may want to prefix with characters, or to add characters to the end of the string, or both.

As an example, suppose you have a text file with lines containing a series of three-digit numbers and you would like to change each of these numbers to **99nnn00**. You can achieve this change with:

```
sed "s/[0-9][0-9]*/99&00/" filename
```

This command says, ‘search for any digit, followed by zero or more digits, and replace this pattern with **99**, followed by the remembered pattern (**&**), followed by **00**’.

MATCHING WITH ^ (CARET)

The caret is one of the two metacharacters used in both regular expressions and filename generation, which, depending on its position in the expression, can match either the character itself or can be used to signify a particular position within the string to be matched. If the caret is in the first position of the expression, it matches the start of the line. If it appears anywhere else, it is the caret character itself, unless, of course, it is used as an exclusion character in square brackets (unlikely to be used in filename generation).

For example, if we wanted to match strings which started with a caret, but were not followed by a lower-case character, we could use the expression:

```
^^[^a-z]
```

We can also escape the caret if we want to match the character itself, no matter where it appears in our expression. In the example above, if our string to be matched included a double caret, we could use:

```
\^^[^a-z]
```

MATCHING WITH $

The **$** is the second of our positional metacharacters that can be used either to match the dollar character or to signify the end of a string or line; the latter interpretation applies only when the **$** is situated at the end of our expression.

For example, **f..k$** would match four-letter words starting with **f** and ending with **k**, situated at the end of our line. To take this one stage further, **^f..k o..$** would match lines which started with a four-letter word beginning with **f**, followed by three-letter word beginning with **o**, followed by the end of the line; **fork out** on a line of its own would thus be matched.

Similarly you can use both of the positional metacharacters to match a blank line using **^$**.

The **$** causes the same problems as brackets when you want to match the character itself. You cannot use commands such as:

```
grep $
grep '$'
grep "$"
grep \$
```

to extract lines from a file containing the dollar character because the shell will interpret the dollar as the end of a line and so **grep** will print out every line in the file. To get round this you must use either the double or quoted backslash pattern we used to match a single bracket, or enclose the dollar with brackets.

Searching for the **$$**, **$***, and **$#** shell variables also causes problems and you must use patterns such as:

```
grep [$][$]
grep [$][*]
grep [$][#]
```

Similarly, if you want to search for a dollar at the end of a line then you must use the pattern **[$]$**.

MATCHING MULTIPLE OCCURRENCES

Repeating with *

The ***** character is probably the most commonly used pattern matching metacharacter, but in regular expressions its importance is not in matching a particular character (apart from itself when escaped or enclosed in brackets) but as an operator to signify zero or more occurrences of the character(s) in the regular expression that precedes it, whether it be a single specified character, one contained in a class of characters surrounded by brackets, or one which is generated from a regular expression. In filename generation, of course, it serves a different function and will match any character apart from a **.** (dot).

We have already seen its usage in our **expr** examples above, where we used **.***; the dot matches any character and ***** produces zero or more of them. This is a catch-all pattern to generate any

number of characters and is usually preceded, or followed, by some regular expression which further defines the specific characters we want to match.

As an example, consider a text file containing the following lines:

```
Candy can can can
Candy can't can can
Candy cannot can can
Candy can not can can
```

What we want to do is extract all the lines with negative statements, which we can do with the following **grep** command:

```
grep "can[ no']*t"
```

In this example the ***** operator checks for multiple occurrences of any of the characters in the brackets, so that the expression will extract **can't**, **cannot**, and **can not**, but will ignore any lines that do not contain at least one **can** followed by zero or more of the characters in the brackets, and then followed by the letter **t**. This expression will thus exclude the positive statement.

As you may have noticed, we have enclosed our expression in double quotes, but it is not essential that **grep** expressions always be quoted. In this particular example the single quote is included in the pattern and unless it is quoted with double quotes the shell will present us with the secondary prompt since it would expect a closing single quote. You only need to quote **grep**'s expressions if they contain characters that could be interpreted by the shell as special characters.

Repeating with ?

The question mark (?) operates much like the *****, but matches zero or one occurrence of the character(s) in the regular expression that precedes it. Do not confuse its usage in regular expressions with that in filename generation where the shell recognizes **?** as a wildcard representing a single character, which is equivalent to the dot in regular expressions.

The question mark is a metacharacter which is available only in a selected number of commands, such as **egrep** and **awk**.

If we wanted to extract lines from the text file we used above, containing either **cannot** or **can not**, then we could use the command:

```
egrep "can[n ]?n"
```

Repeating with +

The plus (**+**) metacharacter behaves in a similar way to the question mark, but matches one or more occurrences of the character(s) in the regular expression that precedes it. It is also available only in a selected number of commands, such as **egrep** and **awk**.

If from our text file we wanted to extract the lines containing **cannot** and **can't**, we could use the command:

```
egrep "can[n']+"
```

MATCHING ALTERNATIVE OCCURRENCES AND GROUP MATCHES

There may be occasions when you want to match one or more expressions by using a single command, and we have already encountered one such construction, which is used extensively in **case** statements. This uses the vertical bar (or pipe symbol if you prefer), **|**, to match any one of a number of regular expressions. This metacharacter is also available in commands such as **egrep** and **awk**.

Using our text file above, we can enter the following command to extract all lines containing **can't**, **cannot** and **can not**:

```
egrep "can[ ]*n|can'"
```

or, the even simpler:

```
egrep "[ ]?no|'"
```

The first regular expression before the vertical bar matches **cannot** and **can not**, and the second expression matches **can't**.

In this extended metacharacter set we also have parentheses to group expressions. They can be used with **egrep** and **awk**, and are also available with **grep** if they are escaped. If we wanted to

match lines containing just **can't** and **cannot**, we could have used the command:

```
egrep "can('|n)"
```

In this particular expression we match **can** followed by either of the characters **'** or **n**, as an alternative to writing **"can'|cann"** as our pattern. As you may have noticed, there are usually multiple ways of writing pattern matching expressions, and whichever you use is immaterial provided that it extracts only the characters you want.

The major difference between using brackets and parentheses in regular expressions, such as in the examples above, is that the square brackets contain individual characters to be matched, whereas parentheses show whole strings that are either expressly stated or are implied from further regular expressions contained within the parentheses; we have already seen examples of this when we used escaped parentheses with the **expr** command to match strings.

To further show that the parentheses match whole strings rather than individual characters, suppose our text file also contained the line **Candy cannt can can**. If we then used the command:

```
egrep "can(not)+"
```

we would match only the line containing **cannot** since we are looking for the **can** characters followed by one or more occurrences of the string **not**, and not the **can** characters followed by one or more occurrences of any of the individual characters, **n**, **o**, or **t**, which would have matched the line containing **cannt** had this been the case.

CLOSURES

There are a pair of metacharacters that allow you to specify the number of times you want a string of characters, or a regular expression, to be repeated. The metacharacters in question are braces, **{ }**. They are available with the **grep** and **sed** commands if they are escaped, not with **awk** at all, and with **egrep** they do

not need to be escaped.

The braces enclose one or two arguments, which is where they get their name, enclosures. The format of an enclosure is:

```
{n,m}
```

where **n**, the minimum number of repeats, and **m**, the maximum number of repeats, are integers between **0** and **256**. The above format will specify any number of occurrences of the preceding regular expression, or single character, or string, between **n** and **m** inclusive. If you specify **{n}** without an **m**, then exactly **n** occurrences are matched. Similarly, with **{n,}** then at least **n** occurrences will be matched.

There are a number of shorthand abbreviations for expressing some enclosures:

- ***** is equivalent to **{0,}**, meaning repeat the preceding pattern zero or more times.
- **+** is equivalent to **{1,}**, meaning repeat the preceding pattern one or more times.
- **?** is equivalent to **{0,1}**, meaning repeat the preceding pattern zero or once only.

In order to show some examples of the use of enclosures, let us assume our text file contains the following lines:

```
Candy can cancan
Candy can't can can
Candy cannot can can
Candy cannt cancan
Candy can not can can
```

To extract lines containing **nn**, we could use:

```
egrep "can{2}"
```

or:

```
egrep "can{2,}"
```

To extract lines containing **cancan**, however, we would use:

```
egrep "(can){2}"
```

This time we have enclosed the string **can** in parentheses so that we look for multiple occurrences of the whole string, rather than just the letter **n**, which we did in the previous example. If we wanted to extract the lines containing **can can**, we cannot use:

```
egrep "(can ){2}"
```

since this will not extract a single line, unless each of our lines contains a space at its end. We also cannot use:

```
egrep "( can){2}"
```

since this will also extract the line containing **can cancan**, which is not what we want. Instead we have to use:

```
egrep "t( can){2}"
```

This searches for the lines containing a **t**, followed by a space, followed by the string **can**, so that it excludes the line containing **cancan**.

Tonto Kowalski
Guru (UAE)

Using tape libraries with AIX

This article focuses on tape libraries from the AIX system administrator's point of control, namely the **tapeutil** and related commands. It discusses some physical aspects like size and location as well as logical aspects like enabling autoloader or partitioning. Some of the most recent changes and enhancements are also mentioned. It does not cover the selection of media type or recommend one or more high-level back-up software products. For ease of use a script is part of the document that adds some SMIT menus to handle the library, especially the medium changer.

TAPE LIBRARIES AND AUTOLOADER

This article concentrates on tape libraries and autoloaders. They

are systems with one or more drives, more tape storage slots than drives, and a roboter that can be accessed from AIX.

Being more exact, there is a difference between a tape library and an autoloader. Usually the autoloader is physically a small library with a single tape drive and robotic. But the handling is much simpler and makes the whole library appear as a single drive with a virtual tape that spans across all physical tapes. Most autoloaders can be switched to the library mode, but the set of tapes written in autoloader mode is not readable in library mode. Some models with LCD displays are in autoloader mode when displaying automatic and in library mode when displaying random access.

The location

Because of the SCSI-bus limitations, the libraries have been close to the systems in the past. For disaster recovery the tapes have been removed from the library. Since there are fibre channel tape drives, the library can be up to 10km away from the systems. Having systems and library in different disaster sections, the removal of tapes may be reduced and therefore the recovery may be speeded up. The environmental stress for the tapes can be reduced as well.

The size

A tape library often seems to provide a huge amount of space. Usually the space is directly related to the data that is stored on disk drives. But most back-up policies need multiple copies of the data, eg for the last week a recovery for each day must be possible. Therefore you will have a number of copies of existing data and a number of copies of already deleted data that needs to go to the library. In addition each tape is usually not filled to the rim, but to an estimated two thirds.

Not considering the long-term back-up tapes that will be removed from the library, the minimum total size of it can be calculated like this:

```
library = 3 / 2 * (data * versions + datadeleted * 2) + 2 *
cartridgesize * hosts
```

where the last term comes in for the boot images of each system. Consider this example; for three ERP systems with 600GB data you will need:

- 600GB in 10 versions (a full week and three for three weeks before).
- 100GB for the hourly incrementally backed up log files.
- Three boot cartridges.

This means 6TB for the database and 200GB log files, resulting in a library size of 9.3TB + 6 cartridges for **mksysbs**. Assume you decided to use LTO2 cartridges with 200GB (uncompressed). Therefore the library should have at least 53 cartridges. There is a benefit of compression and incremental back-ups, but there is also some growth. For the sake of simplicity it is assumed to match in the first year.

The type

There is not enough space here to describe all libraries. The parts of each library are numbered. Mostly each type is numbered consecutively, ie the medium changers, the tape drives, and a fair number of cartridge slots. Optionally there may be a number of so-called mail slots, which may be used to import and export cartridges. The numbers that are given are used by the library control command **tapeutil**.

We do not bother about the exact type of drives because there is a wide variety of proprietary drives, eg AIT from Sony, DLT from Quantum, MagStar from IBM, and many others. There are also a few industry standard technologies, like the Linear Tape Open (LTO), which now is available in its second generation. Tape recording is also divided into helical and linear recording methods. And there are many more differences. However, the technical specifications of the tape drive or library may affect availability of certain functions described below.

Tape devices

There are two types of device we need to consider with tape libraries (excluding those being created for back-up software, eg lbX for TSM). First, there are the tape drive devices, which cannot be distinguished from the stand-alone ones. Second, there is usually one medium changer, which controls the library robotics.

Tape drives

Usually raw magnetic tape drive devices are associated with */dev/rmtX*, with *X* being the number of available drives.

In certain cases one might need to modify the access behaviour. This can be achieved by calling the tape device with its dot extension. As mentioned, the *X* part of the *rmtX.Y* file name specifies which tape drive is used. The first tape drive connected is *rmt0*, the second is *rmt1*, and so on. The *.Y* extension on the file name specifies the tape drive access and writing density as shown in the following chart:

Special File	Retension on Open	Rewind on Close	Density Profile	Unload on Close	Trailer Label
rmt*	no	yes	1	No	No
rmt*.1	no	no	1	No	No
rmt*.2	yes	yes	1	No	No
rmt*.3	yes	no	1	No	No
rmt*.4	no	yes	2	No	No
rmt*.5	no	no	2	No	No
rmt*.6	yes	yes	2	No	No
rmt*.7	yes	no	2	No	No
rmt*.null	yes	no	2	No	No
rmt*.1Ø	no	no	1	No	No
rmt*.2Ø	no	yes	1	Yes	No
rmt*.4Ø	no	yes	2	No	Yes
rmt*.41	no	no	2	No	Yes
rmt*.6Ø	no	yes	2	Yes	Yes

Some of the devices have been available with AIX for a long time, and some of them are not very often referenced. The *rmt*.null* is meant for debugging and developing and there are some similarities with */dev/null*, ie it does not bother any real drives and every action completes successfully. Note: the rmt*.10 bypasses normal close processing and the tape is left at the current

position. With LTO and Magstar the density bit is ignored. More details can be found in IBM *Ultrium Tape Device Drivers Installation and User's Guide*.

Medium changer devices

Mostly the medium changer is accessed through a child device of the first library drive. More precisely, there is a LUN-1 to the same SCSI ID, where the LUN-0 tape drive *rmt0* is connected. In some cases (eg 3494) there is a separate (serial) connection, which is assigned to another device like */dev/rmt0.smc*.

The medium changer accepts commands issued by the ATape driver. The standard interface command in AIX is **tapeutil**. In the case of a serial connection to the 3494, the medium changer is supported by IBM 3494 Enterprise Tape Library Manager Control Program (atldd). The installation is decribed in Chapter 40 of GC35-0154-09, the IBM *TotalStorage Tape Device Drivers Installation and User's Guide*.

The tapeutil command

As already mentioned the **tapeutil** command allows you to control the library functions. **tapeutil** (for ATape driver download see a IBM FTP Server section */storage/devdrvr*) in Version 7.1.5.0 has a lot of subcommands:

```
Usage: tapeutil [-f Device Subcommand [Subcommand ...]]

 General Subcommands:
   devinfo       inquiry [Page]   print "Text"
   reserve       release          reqsense
   reset         logpage "Page"   modepage "Page"
   qrypath       resetpath    disablepath "Primary | Alternate | Number"
   path          checkpath     enablepath "Primary | Alternate | Number"
   tur           vpd              fuser
   passthru      loop [Count]     sleep "Seconds"
   kill

 Medium Changer Subcommands:
   allow         prevent          audit [Address [Count]]
   inventory     mount [Slot]     position "Destination"
   elementinfo   unmount [Slot]   move "Source" "Destination"
   devids                         exchange "Source" "Dest1" "Dest2"
```

```
Tape Subcommands:
  append      bsf [Count]     bsr [Count]
  autoload    eof [Count]     weof [Count]
  noautoload  eofimm [Count]  weofimm [Count]
  compress    fsf [Count]     fsr [Count]
  nocompress  erg             logsense
  load        erase           display "Message"
  mtdevice    rewind          read -d Destination [-c Count]
  qrypos      retension       write -s Source
  seod        status   rtest  [-b Blocksize] [-c Count] [-r Repetition]
  offline     parms    wtest  [-b Blocksize] [-c Count] [-r Repetition]
  rewoffl     sync     rwtest [-b Blocksize] [-c Count] [-r Repetition]
  unload      volid "Name"    setpos [Blockid]
  list        sdp "Number"    chgpart "Number" [Blockid]
  density     idp             qrypart
  sili        prevent         allow
  nosili

Service Aid Subcommands:
  dump [Filename]   fmrtape   resetdrive   ucode "Name"
```

There are some *Installation and Update FAQs* as well as the *Tapeutil Documentation* (Chapter 8) available at IBM.

Generic tape control commands

The most frequently-used commands are those within the *Tape Subcommands* section. Some of those commands are well-known and accessible by the System V standard command mt:

```
weof eof fsf bsf fsr bsr rewind offline rewoffl status
```

After a tape is inserted into a drive you may forward or backward search for files or records, and write an end of file mark. If finished you may rewind and offload by a single or separate commands. Additionally the AIX-specific command **tctl** supports the self-explaining subcommands:

```
erase retension read write reset
```

The remaining subcommands in that section may be used for testing, to modify the behaviour of the drive/library (eg function as autoloader), or are related to barcode labels. Throughout this and all other sections are also commands to improve scripting.

Service Aid and general subcommands

The Service Aid subcommands are used in the case of defects or hardware maintenance, eg firmware updates. The general subcommands provide mostly debugging functionality and commands to manipulate the access to the drives. The reserve/ release command allows users to block a device for exclusive access, eg within a fibre channel. The other subcommands of these sections are not considered here in detail.

Medium Changer Subcommands

Most important for handling cartridges are the Medium Changer Subcommands. While **move** moves a single cartridge from one slot to another, **exchange** additionally moves a cartridge present in the target slot to another space. The **mount** and **unmount** commands allow you to move a tape between a slot and a drive. With **position** the access times may be reduced by having the accessor lurking in front of the given slot.

The library inventory

Elementinfo gives a short overview of the components within the library, while **inventory** lists all the details of those elements. Usually part of the information is not available when connecting the library to the power. It needs to be collected with the audit command, which checks in which slots cartridges are and with what labels. And for a big library this may take as long as you would expect (or even longer!).

The subcommand **inventory** gives you a lot of information. When moving the tapes around it can be helpful to reduce the amount of data to the most relevant parts. The following script picks only the slots where cartridges are present. After the slot number it displays from where the cartridge comes, what type the slot is, and what label was read.

```
#!/usr/bin/sh
#
# command:      showlabel <smc#>
#
# example:      # showlabel smcØ
```

```
#               ID [From] Type        <Label>
#               32 [32] Slot address <F2Ø6C38>
#               33 [33] Slot address <F2Ø6C39>
#               [...]
if [ $# -lt 1 ]
then
    exit -1
fi
echo "ID [From] Type        <Label>"
(tapeutil -f /dev/$1 inventory 2>/dev/null && echo) |\lineawk '
        if(NR>1)
           if(index($Ø," ")>1)
               address=$NF;
                type=$1 " " $2
             else
            if(index($Ø," ")==1)
               if($1=="Source") former=$NF;
                if($2=="Tag") label=$4
                if($2=="Present") media=$4
             else
            if(media=="Yes")
            printf"%d [%2s] %s <%s>",address,former,type,label

    '
```

Inventory does not care for the AIX drive naming. The only way of mapping drives and slot numbers is by looking at the SCSI IDs and LUNs. Moreover, usually it is more interesting to have a view of the cartridges within the drives than the drive itself. The following script presents the AIX name together with the bus address, the slot number, and the cartridge label:

```
#!/usr/bin/sh
#
# command:      showdrive <rmt#>
#
# examples:     # showdrive rmtØ
#               rmtØ (LUN Ø,Ø) 16 <-- 66 <F2Ø6C38>
#               # showdrive rmt1
#               rmt1 (LUN 1,Ø) 17     empty
if [ $# -lt 1 ]
then
    exit -1
fi
DRIVE=$1
echo "$DRIVE ("
LUN=$(lscfg -l $DRIVE | awk 'if(NR==3)split($2,a,",");printf"Logical
Unit.* %s",a[2]')
```

```
SID=$(lscfg -l $DRIVE | awk 'if(NR==3)split($2,a,",");c=split(a[1],b,"-
");printf"SCSI.* %s",b[c]')
tapeutil -f /dev/$DRIVE inventory 2>/dev/null |\linegrep -p "$LUN"
|\linegrep -p "$SID" |\lineawk '
    if(index($0," ")>1)
       address=$NF;
        type=$1 " " $2
     else
    if(index($0," ")==1)
       if($1=="Source") former=$NF;
        if($2=="Tag") label=$4
        if($1=="SCSI") scsi=$NF
        if($1=="Logical") lun=$NF
        if($2=="Present") media=$4
     else
    if(media=="Yes")
        printf"LUN %d,%d) %d <-- %2s <%s>",scsi,lun,address,former,label
    else
        printf"LUN %d,%d) %d     empty",scsi,lun,address
'
```

Positioning the tape

Frequently with large cartridges, like the Ultriums, one wants to add another archive after the last record on tape. Using the *rmt*.1* (or *rmt*.10*) device you can avoid moving the tape after the command has completed:

```
tapeutil -f/dev/rmt0.1 seod
```

This command spaces to the end of data and leaves the drive at that position. If you do not retension or rewind on open or before, your data goes to the end of the tape.

With the **setpos** and the **qrypos** subcommands, you can position the tape at an arbitrary block. Either the blockid must be given or the tape must be previously treated with **qrypos**, which causes the drive to remember the blockid.

Tape cartridges

According to the IBM recommendations *Tape Backup – Handling tape cartridges* (IBM document ID: MIGR-39484) and *Tape Backup – Media operation and care considerations* (IBM document ID: VLAR-42XRPP) you should obey the rules for

media handling that are distributed with the media, ie avoid any kind of radiation or other environmental threats, particularly try to keep the media protection lid shut to reduce tape contact to a minimum.

It is recommended that you store the cartridges vertically and use barcode labels. The barcode label is meant to simplify access to the reference information. If no barcode label is used, one should use the reference information on the label itself. This should specify the cartridges' characteristics. The label should identify who stored what information, when, and in which format. The software should export the reference information to the cartridges as well.

Tape drive cleaning

The environment for tape libraries and the drives within them must be clean. Otherwise the commonly assumed lifetimes of tapes and drives will not be reached. Moreover this can result in data loss.

But even within a clean environment there is still some dirt left within the drives when working with the tapes. Therefore the drives need to be cleaned. There was a time when this had to happen on a regular basis, eg 8mm with 2.3GB capacity had to be cleaned every month or after 30 hours of operation. Newer tape drives indicate their need for cleaning on their control panel. Within most libraries there are functions that automatically load the cleaning cartridges to allow proper function without manual intervention.

A tape drive needs cleaning when one of the following conditions occurs:

- Its LCD/LED display indicates cleaning.
- The attached system's error report indicates cleaning.
- Certain media changes are executed (eg AME after MP in 8mm drives).

- Operating hours exceeded (usually only the withdrawn 3570 and older drives without service indicator).

The more modern the drives are, the less the tape touches the head and the positioning coils. In consequence there is less need for cleaning.

Cleaning cartridges

Every cleaning cartridge has its lifetime, ie it may be used for a limited time only. After that time it should not be reused for any drive. Most cleaning tapes have a lifetime of between 12 and 50 cleaning cycles. Details for non-LTO and non-Magstar are mentioned in the corresponding chapters of the IBM *Tape Help Package*, which also gives the correct order numbers.

Cleaning the drive

In some libraries, like the Magstar MP series ones, its controller counts the times the cleaning cartridge is used. If there is no library support, the usage times must be counted manually.

Many recent drives support a kind of self-administration with respect to their cleaning needs. Using the SCSI bus to read the drives' memory, the **utape** command allows you to find out how many hours of operation have passed since the last cleaning cycle:

```
/usr/lpp/diagnostics/bin/utape -vcnd /dev/rmtØ
```

The time in hours since the last usage of the cleaning tape is reported. The **utape** program is part of the ATape driver distribution and is described as performing the following actions:

```
Usage: utape [-h] | [-d <device>] [-n|-t|-l]
       utape -c [-v] -d <device>  -n | -l | -t -f [<filename>] |-D
```

where:

- -c – command line.
- -v – verbose mode.
- -h – usage statement.

- -d – device name.
- -n – time since last cleaned.
- -t – trace table.
- -l – logsense data.
- -f – write to file.
- -D – write to DOS diskette.

The command is also documented in almost all pSeries user guides, eg for the model F80. It can be used to retrieve the log sense or trace information to floppy or file. Both ways may help to track down errors, but are less useful in day-to-day business.

TAPE PERFORMANCE

While the IBM *Tape Help Package* mentions a lot of influencers on tape performance, we take some as given, like the CPU's power or bus speeds. Nevertheless, there is still a lot that we can do to identify good performance.

Topology and throughput

Check the topology for theoretical and practical throughput. Convert all theoretical values to MBps (or MB/s), since it is the usual unit for measuring the speed of tape drives. GBph (Gbyte per hour) may be converted to MBps by dividing by 3.6, while Gbps (gigabit per second) may be converted to MBps by multiplying the number in front of the units by 125. This means if you can back-up an SSA-Array with 216GB within 5 hours, your tape performance is 43.2GBph, which is 43.2/3.6MBps=12MBps.

Topology means how the data source and the tape drives are connected. For each transfer you need a path. With dedicated SCSI this is rarely a problem. With fibre channel you need to watch out. Like on a motorway, it does not help having four lanes at each end when in between the traffic is narrowed to a single lane.

As a rule of thumb, try to connect each modern tape drive on a

separate path and if unavoidable use a maximum of two drives per path, where the path and the drive generation should match.

Compression

For performance reasons, it is important to choose the right location for compression. Most drives perform best when always having a filled buffer. Therefore it is necessary to deliver a continuous and uninterrupted stream that is capable of sending faster than 50% of the nominal drive performance. LTO2, for example allows adaptive speed control between 18 and 35MBps.

The following example does some maths on compression. A 2Gbps FC link allows a theoretical performance of 2*125MBps=250MBps (practical throughput is less, see documentation). Having 4 LTO-2 drives on a single 2Gbps path it would be sufficient to send already compressed data to those drives, meaning a maximum throughput of 4 drives at 35MBps/drive or 140MBps in total. But uncompressed data may double the ratio of path-to-drive demands, ie you have to send with 70MBps/drive to keep each drive streaming with 35MBps, since compression reduces the amount of data before it's put on the tape after receiving it at the drive. In situations with extremely compressible data, a throughput of up to 120MBps per drive is possible. In conclusion, it is better to have a dedicated 2Gbps path to a maximum of two LTO-2 drives if full performance is required even with uncompressed data. (Please note that you can assume most video, audio, and picture data will already be compressed.)

Another rule of thumb is to compress the data at the source system if your weakest component(s) are the cabling (aka networks), and to rely on the hardware compression in the drives if your weakest component(s) belong to the systems.

Testing

Testing is not that easy when talking about compression. Most people use a **cat /dev/zero** or the AIX-specific command **lptest** to generate a data stream. But be aware that both streams may

be pretty well compressed. These are not good input for testing a tape drive with compression enabled. The following table shows that even the oldest compression programs like **pack** reduce a stream of zeros to an eighth of its original size:

Size in Bytes	*Command*	*Filename*
524288	dd if=/dev/zeroof=/tmp/zero bs=32k count=16	/tmp/zero
65546	pack /tmp/zero	/tmp/zero.z
1283	compress /tmp/zero	/tmp/zero.Z
548	gzip /tmp/zero	/tmp/zero.gz
45	bzip2 /tmp/zero	/tmp/zero.bz2
204800	lptest 1023 >/tmp/lptest	/tmp/lptest
170713	pack /tmp/lptest	/tmp/lptest.z
9745	compress /tmp/lptest	/tmp/lptest.Z
1480	gzip /tmp/lptest	/tmp/lptest.gz
1794	bzip2 /tmp/lptest	/tmp/lptest.bz2

I do recommend some MP3 audio data or, even better, some MPEG4 or DIVX video data, which was tested with bzip2 compression before. In principle GIF and JPEG data would be also a good choice, but you are going to need a lot of them, since they should not repeat within one buffer filling at least.

Buffer size

When thinking about block size, keep two things in mind:

- The device driver.
- The application.

The operating system uses a device driver to communicate with a device. Therefore we need to determine the blocksize currently set for the tape drive by an AIX system command:

```
# lsattr -El rmt0  # (Write down this blocksize)
```

Setting the blocksize will be done with the following command:

```
# chdev -a block_size=32768 -l rmt0
```

One may use smit tape and its sub menus to achieve the same result. LTO Generation 2 supports buffer size of 64Kbyte (ie **block[lowbar]size=65536**).

Please be aware that there can be several difficulties when

having the wrong blocksize. Using blocksize 0 allows you to read all kinds of data but performance might be quite bad. Booting from tape might need a certain blocksize. Please verify that your mksysb tapes will work if you change the block size.

For calculating the block size with which a tape was recorded, proceed as follows:

- Write down this blocksize:

  ```
  lsattr -El rmtX
  ```

- Set the tape drive's blocksize to 0 (if not already at 0):

  ```
  chdev -a block[lowbar]size=0 -l rmtX
  ```

- Determine the blocksize the tape was written at:

  ```
  dd if=/dev/rmtX bs=1024k count=1 [verbar] wc -c
  ```

- Set the blocksize for the tape:

  ```
  chdev -a block[lowbar]size=... -l rmtX
  ```

- Process the tape.

- Set the blocksize to the recorded original or the new desired value:

  ```
  chdev -a block[lowbar]size=... -l rmtX
  ```

In practice (native AIX back-up commands)

The most common native AIX applications for back-up are discussed here. Specialized back-up applications like Tivoli Storage Manager or Legato Networker are not discussed. See their documentation for details.

For modern tapes, a block size of 32KB should be a minimum consideration when writing to tape. This is the block size we'll use. If you are not familiar with the commands, please consult the manual to avoid surprises, since some commands might exhibit some strange behaviour; eg **cpio** has a file limit of 2GB, and **pax** may restrict path components to less than 100 characters.

The following back-up commands are invoked without

compression because they are described for local usage. The situation changes when the target tape drive is located at a remote site and must be accessed via a LAN (or, even worse, through a WAN). There is some reference in the chapter about back-up and restore in the AIX *Base Administration Guide* on how to modify the commands discussed here to write data to a remote drive.

When the bandwidth to the remote system is or might be a bottleneck so that the drive lacks data to write continously, then the data should be compressed before sending it across the network. Therefore, add a local pipe through a compression command before calling the remote operation, eg with **rsh**. An example is the following back-up/restore command pair:

```
tar -cdf- ...pathname... | rsh remotehost "dd of=/dev/rmtØ ..."
rsh remotehost "dd if=/dev/rmtØ ..." | tar -xvpf- pathname
```

which should be modified in the following way to avoid a bottleneck:

```
tar -cf- ...pathname... | gzip -c | rsh remotehost "dd of=/dev/rmtØ ..."
rsh remotehost "dd if=/dev/rmtØ ..." | gzip -cd | tar -xvpf- pathname
```

Other commands may be modified in a similar way. Also the compression command may be varied. Particularly on older systems, you should pay attention to how fast the compression commands work, since they might be a bottleneck themselves.

Back-up with backup

The blocksize option for **backup** is set by using the option **-b *number***. The meaning of *number* depends on the back-up method:

- The flag **-level** (typically used with **-u**) results in a back-up by **inode**, where *number* specifies the number of 1024-byte blocks for a single output operation. The default value for *number* is 32, which means 32,768 bytes per write.
- The flag *-i* results in a back-up by name, where *number* specifies the number of 512-byte blocks for a single output

operation. The default value for *number* is 100, which means 51,200 bytes per write.

It is up to you to ensure that the write size is an even multiple of the tape's physical block size.

The **backup** command in **i-node** mode is equivalent to the dump (not AIX!) command known from other Unixes. It should be applied only to unmounted or read-only file systems to avoid inconsistencies. The by-name mode is more similar to **tar** or **cpio** and is used by **mksysb** and **savevg**.

SAVE Volume Group savevg and MaKe SYStem Backup mksysb

The command **mksysb** writes a **savevg** for the rootvg to tape, after preloading it with a boot code that allows easy restoral. The **mksysb** cannot be used with an autoloader device. Autoloaders can be temporarily set to the normal library mode by using **tapeutil -f /dev/rmt0 noautoload write -s backup.tar**, but this is not recommended for **mksysb**.

While **mksysb** is documented to behave like **backup** as far as it concerns the block size, the **savevg** command seems to adapt to the physical block size. The option handling for changing block size is identical to the **backup** command.

Tape ARchiver tar

The **tar** command originally supports a default buffer size of 10K, which is described in the man page as maximum back-up size. Some man pages apparently describe the default block size of **tar** as 512K (which is wrong).

The only way to increase the block size with **tar**, when writing to tape, is to use the cluster block option **-N**. Using **-N64** results in 32KB written to tape in a single operation. Since **tar** is not able to determine the large block sizes automatically, it is highly recommended to use this option for all operations including read.

CoPy from Input to Output cpio

The **cpio** command is a wonderful tool if you prefer to select the

files to back-up with a non-trivial **find** command. It supports a default blocksize of 512 bytes when using the option **-B**. Mutually exclusive with it is the option **-C**. With **-C64** you will achieve 32K block sizes.

When writing to tape, the block size must be equal or a multiple of physical block size. When reading from tape, the block size should not be larger than the physical block size.

Portable Archive eXchanger pax

Apparently, **pax** was created to reconcile the **tar** and the **cpio** fans (although I don't know many people who use it). It allows you to specify the output format with **-x** either to be:

- **ustar** with the tar default block size of 10K.
- **cpio** defaults to the block size of 20K.

Both allow the blocksize to be varied between one and 64 times 512 bytes, allowing a maximum of 32K. The syntax is **-b 32k** or **-b 64b**.

Data Dump dd

The **dd** command is very flexible and may be used to convert any input that comes from standard input to a tape drive with a specified blocksize. **dd if=zz of=/dev/rmt0 bs=32k** takes the input from *zz* to the tape drive *rmt0* using a block size of 32K.

Doing back-ups

Conventional back-up

As described in the AIX *System Management Guide* in the chapter *Backup and Restoring Information*, user data can be backed up incrementally by **i-node** (here level 5):

```
# DO THE BACKUP
# close the file system
umount /filesystem
# check the file system
fsck /filesystem
```

```
# backup the file system by i-node incrementally
# with a block size of 32k bytes (not necessary for 32k)
backup -b32 -5 -uf/dev/rmtØ /filesystem
# verify that the backup is readable
restore -tvf /dev/rmtØ
# BACKUP COMPLETED
```

For a regular back-up it is usually more appropriate to do a back-up by name, which also allows you to restore single files. Here is just a simple example to back-up some directory trees. Since there is no need to unmount the file system for this variant the previous script would consist of a single line.

If you are using a library with at least seven tapes, the following script can be used to do a daily back-up and to overwrite each tape after one week. Save these lines to a file and call this script daily with cron. The seven slots starting with firstslot are used for this back-up:

```
# prepare the tape slot numbering
tapedrive=/dev/rmtØ
firstslot=32
weekday=$(date +%w)
storeslot=$(expr $firstslot + $weekday)
# preparing the tape drive
/usr/bin/tapeutil -f /dev/smcØ mount $storeslot
# erase last weeks backup and rewind the tape
/usr/bin/tapeutil -f $tapedrive erase rewind
# DO THE BACKUP
/usr/sbin/backup -if$tapedrive.1 /home/some
/usr/sbin/backup -if$tapedrive.1 /home/further
# BACKUP COMPLETED
# cleaning up; unloading the tape drive
/usr/bin/tapeutil -f /dev/smcØ unmount $storeslot
```

Point-in-time back-up

New with AIX 5.2 are the commands **backsnap** and **snapshot**. Using the **backsnap** command allows you to redirect a point-in-time copy of a JFS2 filesystem directly to tape. Here's an example of how to back-up a user's home directory exactly the way it looks when the command is executed (no matter whether the user erases files while the back-up is written to tape):

```
backsnap -R -m /tmp/snapshot/myuser -s size=16M -i -b 64 -f /dev/rmtØ
```

This command creates a logical volume with a size of 16 megabytes and then creates a snapshot for the */home/myuser* file system on the newly-created logical volume. From that copied data, a back-up is taken to */dev/rmt0*. The **-R** means remove this snapshot after the back-up is complete. The sequence **-i -b 64** tells the back-up to make a back-up by name and use 32K instead of 51.2K block size.

CHECK TAPES

The file system check **fsck** for file systems is a common action. A tape should be checked as well. There are some commands that may be helpful.

Tape check tapechk

The **tapechk** command does a very simple job by rewinding and reading a specified number of files, eg **tapechk 3** reads the specified three files. The variable TAPE must be set to use a drive other than */dev/rmt0*.

Tape copy tcopy

The **tcopy** command was originally designed to copy from one tape to another. If called with two arguments that are tape drives, eg **tcopy /dev/rmt0 /dev/rmt3,** data is read from the first and written to the second. If there is only one argument, all back-up images with their size are displayed for the specified tape drive.

The following test runs show example output, when it is called with a single loaded tape drive as an argument. First, a drive with a good cartridge:

```
# tcopy /dev/rmtØ
 tcopy: Tape File: 1; Records: 1 to 3; Size: 8Ø.
 tcopy: File: 1; End of File after: 3 Records, 24Ø Bytes.
 tcopy: Tape File: 2; Records: 1 to 4Ø63; Size: 262144.
 tcopy: Tape File: 2; Record: 4Ø64; Size 189518.
 tcopy: File: 2; End of File after: 4Ø64 Records, 1Ø65Ø859Ø Bytes.
 tcopy: Tape File: 3; Records: 1 to 2; Size: 8Ø.
 tcopy: File: 3; End of File after: 2 Records, 16Ø Bytes.
 read: There is an input or output error.
```

```
tcopy: The end of the tape is reached.
tcopy: The total tape length is 1065280990 bytes.
```

Now look at the output for a cartridge that is damaged. Please note that there is always a read error before the end of tthe tape is reached. The message printed is either the general message as in the above example or the previous error, if any:

```
# tcopy /dev/rmt0
tcopy: Tape File: 1; Records: 1 to 3; Size: 80.
tcopy: File: 1; End of File after: 3 Records, 240 Bytes.
tcopy: Tape File: 2; Records: 1 to 1552; Size: 262144.
[...]
tcopy: Tape File: 2; Record: 50334; Size 66337.
read: The media surface is damaged.
tcopy: Tape File: 2; Records: 50335 to 51122; Size: 262144.
tcopy: File: 2; End of File after: 51122 Records, 13365114800 Bytes.
read: The media surface is damaged.
tcopy: The end of the tape is reached.
tcopy: The total tape length is 13365115040 bytes.
```

TAPEUTIL.SH

```
#!/usr/bin/sh
################################################################################
## Description:
##      This shell script adds a tapeutil management option to your
##      SMIT's top level menu APPLICATIONS which is initially empty.
##
##      You do not need to provide parameters.
################################################################################
NAME=tapeutil
################################################################################
# Inform the installer what is going to happen
################################################################################
cat <<***
      You are currently executing a script to add
      a \'\'tapeutil" management to your SMIT menu
      called "Applications" (top level) for one of
      your applications.  Please note that it is
      verified to find a correct installation of
      \'\'tapeutil", but not the existence of a
      tape library.
***
################################################################################
# Checking for a correctly installed 'tapeutil' before writing stanza
# file
################################################################################
```

```
alias atapeinfo="lslpp -ch Atape.driver | tail -1"
if [ ! $(atapeinfo|cut -d: -f5) = "COMMIT" ]
then
      if [ ! $(atapeinfo|cut -d: -f5) = "APPLY" ]
      then
            echo "      Please commit or replace Atape.driver first."
            exit -1
      fi
      echo "      You do not have an Atape.driver installed."
      if [ -x /usr/local/bin/lwp-request ]
      then
            lwp-request -P "http://proxy" \
            ftp://index.storsys.ibm.com/devdrvr/AIX/Atape.'*'.bin \
            >/usr/sys/inst.images/Atape.recent.bin
            inutoc /usr/sys/inst.images
    echo " A Recent version was downloaded to file:/usr/sys/inst.images"
      else
    echo " Downloaded it from ftp://index.storsys.ibm.com/devdrvr/AIX/"
      fi
      exit -1
fi
if [ ! $(atapeinfo|cut -d: -f6) = "COMPLETE" ]
then
      echo "      Your Atape.driver installation is not complete."
      exit -1
fi
echo "      You are using Atape.driver $(atapeinfo|cut -d: -f3)."
##############################################################################
# Are we allowed to do it ...
##############################################################################
if [ $(id -u) -ne Ø ]
then
      # USER is not root; do not execute the script
      echo "  Gain root priviledges before executing this script."
      exit -1
fi
ODMDIR=/usr/lib/objrepos
SEQN=$(date +%Y%m%d%H%M%S)
echo "\t\t... adding your menu now"
##############################################################################
# Writing stanza file and add it to ODM in file:/usr/lib/objrepos/
##############################################################################
cat <<*** >/tmp/smitty.add
sm_menu_opt:
      id_seq_num = "1998"
      id = "apps"
      next_id = "tapeutil"
      text = "Tape Library Management Utility"
      text_msg_file = ""
      text_msg_set = Ø
```

```
      text_msg_id = Ø
      next_type = "m"
      alias = ""
      help_msg_id = ""
      help_msg_loc = ""
      help_msg_base = ""
      help_msg_book = ""
sm_menu_opt:
      id_seq_num = "1999"
      id = "$NAME"
      next_id = "smc"
      text = "Medium Changer List Commands"
      text_msg_file = ""
      text_msg_set = Ø
      text_msg_id = Ø
      next_type = "m"
      alias = ""
      help_msg_id = ""
      help_msg_loc = ""
      help_msg_base = ""
      help_msg_book = ""
```

Editor's note: this article will be concluded next month.

Andreas Neuper
PROFI Engineering Systems (Germany)

Opsware has announced its Automated Script Execution (ASE) Subsystem, a new capability for executing scripts across hundreds or thousands of servers. It's designed to simplify the way IT administrators make everyday changes such as auditing configurations, changing passwords, or reconfiguring network settings on servers.

ASE, among other things, lets IT administrators use Opsware APIs to retrieve information from the Opsware platform, enabling a single script to adapt itself dynamically to many different environments and situations.

It incorporates an audit trail, role-based access control, encrypted communications, and digital signatures for all scripts, and it can simultaneously execute scripts from one central location across thousands of servers.

Supported platforms include AIX, HP-UX, Linux, Solaris, and Windows.

For further information contact:
Opsware, 599 N Mathilda Avenue, Sunnyvale, CA 94085, USA.
Tel: (408) 744 7300.
URL: http://www.opsware.com/software/index.htm.

* * *

Candle has introduced PathWAI Secure for WebSphere MQ, which expands the protection of information across the WebSphere MQ environment by combining existing security and management applications with encryption software from RSA.

Increased authentication verifies the identities of message senders and recipients, PKI support strengthens security, and there's an expanded ability to validate that data transmissions and message archives that have not been altered.

The software supplements the user authorization capabilities of various external security programs, such as RACF, ACF2, and Top Secret on OS/390, as well as operating system security tools for Unix and Windows systems.

The software supports AIX, OS/390, z/OS, AS/400, HP-UX, Solaris, and Windows NT, 2000, and XP.

For further information contact:
Candle, 201 N Douglas St, El Segundo, CA 90245, USA.
Tel: (310) 829 5800.
URL: http://www.candle.com/www1/cnd/portal/CNDportal_Channel_Master/0,2258,2683_1896860,00.html.

* * *

BMC has announced new Patrol tools for SAP application management, which are designed to propagate relevant third-party availability and performance data into the Computer Center Management System (CCMS) of the SAP Basis Release 4.6.

User response time breakdown and alerting on response time breaches are provided, as well as support for PeopleSoft 8.8 and PeopleTools 8.42 and for AIX 5.1 and Oracle 9*i* platforms.

For further information contact:
BMC, 2101 CityWest Blvd, Houston, TX 77042, USA.
Tel: (713) 918 8800.
URL: http://www.bmc.com/products/documents/09/92/20992/20992/index.htm.