80

AIX update

June 2002

In this issue

3 Using and referencing variables
13 Core dumps in AIX 5L
17 Understanding the head and tail commands
25 Removing users from a system
29 Variables and the environment
38 Advanced features of EMC PowerPath software
48 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.

Using and referencing variables

REFERENCING VARIABLES

If you try to reference a variable in a script (or from the command line) that has not been assigned a value, the shell will effectively ignore the variable and run the resultant command, which may produce errors depending on the syntax required for the command. If you wish, you can deliberately assign the null string to a variable. For example:

```
obscure=
```

Referencing the value of a shell variable that has been assigned the null string has the same effect as referencing a variable that does not exist.

There are a number of *string operators*, using a curly-bracket syntax, which allow you to manipulate the values of variables. Some of these operators, for example, allow you to specify a default or back-up value to be used in case a variable does not exist, or has a *null* value, while others are used to extract characters from variables containing strings that match prescribed patterns.

They are very useful constructions to know, since they allow users who may not yet be experienced shell programmers to manipulate variables without having to write complex programs, although they can be difficult to remember in detail. These string operators are discussed in the following sections.

${variable:-string}

If an expression such as:

```
${variable:-string}
```

appears on a command line, the expression will be replaced with the current value of *variable*, if it exists. If it does not exist, or has been assigned a null value, the reference will be replaced with the *string* instead.

An example of the use of this type of parameter substitution is:

```
TERM=${TERM:-vt100}
```

You can try the above type of parameter substitution by first assigning a value to the variable **greeting**, as follows:

```
greeting=hello
```

Now enter:

```
print ${greeting:-bye}
```

and notice that **hello** is displayed. If you display the value of greeting with the **set** command, you will see that it is still **hello**. If you now assign the null string to **greeting**, and enter the same command, this time **bye** is displayed, but on entering **set** you will see that **greeting** still has the null value.

If you now **unset** greeting, so that it no longer exists, and enter:

```
print ${greeting:-bye}
```

then **bye** will again be displayed, and **set** will show that **greeting** still does not exist.

${variable:=string}

Now try a modification of the command, but first **unset greeting** and then enter:

```
print ${greeting:=bye}
```

This will again display **bye**, but on using **set** you will see that the variable has now been created and set to the value **bye**.

If you now set **greeting** to null and enter the above command again, **bye** will again be displayed, and **set** will also reveal that **greeting** has now been given the value **bye**.

If **greeting** is set to **hello**, then on entering:

```
print ${greeting:=bye}
```

the value **hello** will be displayed and **greeting** will be unchanged.

${variable:+string}

If **greeting** does not exist, or is equal to the null string, then the further

variation of the command:

```
print ${greeting:+hi}
```

will display the null string. If **greeting** is equal to null, or is not set, then **set** will reveal that its value is still null.

If **greeting** already equals **hello**, then the above command will display the string **hi**, but **set** will reveal that **greeting** still retains its original value.

${variable:?string}

This type of string operator allows you to print a message, defined by the string, if the variable does not exist or has a null value. It will also cause any script in which it is contained to abort. If the variable does have a value, then this value will be printed.

We can now create a simple script that can make use of this string operator. Let us assume that we want our script to accept a command line argument, which the shell refers to as **$1** (this type of input for shell scripts will be discussed in greater detail in a future article**)**.

Create the script called **args**:

```
#!/bin/ksh
print ${1:?No argument} >/dev/null
print Have an argument
```

If we make the script executable and then run it without an argument, it will send the following output to standard error, and then exit:

```
args: 1: No argument
```

If we run the script with an argument, then only the message, "**Have an argument**", will be displayed. This is because we have redirected standard output for the first **print** statement to */dev/null*. If this redirection was not in place then we would also have the value of the argument printed first, and we have decided (in this particular case) that this is not desirable.

${variable#string}

It is also possible to delete parts of variables when they are referenced. If you set **greeting** to **hello**, and then enter the command:

```
print ${greeting#hel}
```

the output from this substitution will be **lo**.

This type of substitution tries to match the pattern following the # with the beginning of the variable, and if it is a perfect match then the matching portion is deleted from the output; the value in memory is unaffected. If there is no match, the value of the variable is substituted.

The characters that follow the # can be a simple string, or they can be any complex pattern matching construction. For example:

```
print ${greeting#[a-z]??}
```

will also print out the characters **lo**.

${variable%string}

A similar form of the above command is:

```
print ${greeting%lo}
```

which will produce the output **hel**. This string operator performs its matching with the end of the variable.

${variable:-$(command)}

Another type of substitution uses parentheses. Whenever you have a command of the type:

```
(cd;ls)
```

the list of commands contained within the parentheses are executed in a sub-shell.

If you use a command of the type:

```
print ${dir:-$(pwd)}
```

pwd will be executed only if **dir** is not set or is null. This type of substitution can also be used when the variable references use the = and + constructions.

${#variable}

It is also possible to display the length of a variable by using:

```
print ${#variable}
```

and if instead of *variable* you use ***** or **@**, then the number of positional parameters is substituted.

SUMMARY

The differences between the output from some of the above string operators is not easy to remember, and it is easy to forget these differences when you use the construction infrequently, particularly when a variable has not been set, or has a null value. To assist you I have created a table which summarizes the differences.

If the command entered is of the form:

```
print ${variable:= or - or + newstring}
```

then the actual values of *variable* stored in memory (displayed by using the **set** command), and the output displayed by **print**, are shown in Figure 1.

Current value	*Value shown by set (stored in memory)*		
	=	-	+
oldstring	oldstring	oldstring	oldstring
not set	newstring	not set	not set
null	newstring	null	null

Current value	*Output from print*		
	=	-	+
oldstring	oldstring	oldstring	newstring
not set	newstring	newstring	null
null	newstring	newstring	null

Figure 1: Differences table

VARIABLE ARRAYS

The Korn shell supports one-dimensional arrays with subscripts (or indices) in the range from **0** to **511**. You can subscript a variable by using **[ ]**. For example:

```
var[0]=000
var[1]=111
var[7]=777
```

Arrays do not need to be declared before using them. Any reference to a parameter array with a valid subscript is acceptable, and the array element is created when it is first assigned.

You do not need to assign values to variables in a specific order, or indeed at all. For example, you could create a variable:

```
planet[5]=jupiter
```

without first having to declare or define **planet[0]**, **planet[1]**, and so on. In fact you need never create them.

There is a further way in which you can assign an initial set of values to the elements of an array by using the **set** command:

```
set -A planet mercury venus earth mars . . .
```

which will set **planet[0]** to **mercury**, **planet[1]** to **venus**, and so on. If for some reason you did not want the first real element of your array to start at index **0**, but **1** instead, then you could assign the **0**th element a null string, **“”**, so that the elements of your array would now be in an order you wanted.

You can reference an individual variable in an array by using:

```
print ${planet[1]}
```

and if you do not use a subscript, this is the same as referencing the variable **planet[0]**.

If you use a command such as:

```
print ${planet[*]}
```

this will display all the elements of the array, separated by spaces. Similarly you can determine how many elements in an array have currently been set with:

```
print ${#planet[*]}
```

To unset all variables in an array you can use the command:

```
unset planet
```

VARIABLE SUBSTITUTION

There may be occasions when you want to assign a value to a variable, and then assign this variable to a second variable. On using the **print** command to display the contents of the second variable, the expected substitution for the first variable might not necessarily be made.

For example, if you make the following assignment:

```
DIR='$PWD'
```

and then enter:

```
print $DIR
```

the output of this command will be the characters **$PWD**, which might not be what you intended.

To overcome this you can use the built-in shell command, **eval**, so that if you enter:

```
eval print $DIR
```

this will produce output of the type:

```
/home/fred
```

The **eval** command should be used when 'hidden' variable substitutions may cause conflicts. It has the same effect as forcing the shell to scan the command line twice before executing the specified command.

In the above example, **$PWD** was enclosed in single quotation marks, which prevented the usual interpretation of the **$** metacharacter when the **print** command was used on its own. With **eval**, however, the **print** command is executed after all its arguments have been expanded by the shell.

eval is also useful for interpreting variable names consisting of two parts (or more), each of which may themselves have different values, and when joined together they produce a third variable whose value you may want to extract.

For example, we may have a number of volume groups and we want to display various attributes for each volume group, such as the total size, number of PPs, auto-varyon value, and so on. The first part of the variable name would contain the volume group name, **$VG**, and the

second part would be an attribute, **$ATTR**. We would then use nested loops containing **eval** statements (with carefully constructed syntax) to display the values of our third variables, which may be in the format **${VG}_$ATTR**.

Since you have not yet been introduced to nested loops, let us consider a much simpler example where only one part of the variable name is required to be changed.

You will remember that in *Examples of shell scripts*, *AIX Update*, Issue 78, April 2002, we created a script called **vgsizes**. Let us suppose that we also want to print the number of logical volumes in each volume group. At present we have hard-coded **rootvg** into the script, but later we hope to modify the script to be able to generate a list of all volume groups varied on, and then use loops to display the number of logical volumes for each.

Modify the script so that it now looks like the following:

```
#!/bin/ksh
VG=rootvg
SOURCE=vgs
TOTAL=$(grep -p $VG $SOURCE | grep "TOTAL PPs" | cut -f2 -d "(" | tr '
' '\t' | cut -f1)
FREE=$(grep -p $VG $SOURCE | grep "FREE PPs" | cut -f2 -d "(" | tr ' '
'\t' | cut -f1)
eval ${VG}_LVNUM=$(lsvg -l $VG | tail +3 | wc -l | tr -d " ")

printf "%-20s %-15s %-15s\n" "Volume Group" "Total Size" "Free Space"
printf "%-20s %-15s %-15s\n" $VG "$TOTAL MB" "$FREE MB"
print ${VG}_LVNUM
eval print Number of LVs in $VG = '$'${VG}_LVNUM
```

The first **eval** command again uses command substitution and merely counts the number of logical volumes in the volume group. The **tail** command extracts only the output lines from line 3 onwards, and the **tr** command removes all spaces in the output from **wc**.

Note that we have enclosed **VG** in braces to ensure that its value is extracted before appending the remaining characters to create the new variable name. If we had not used braces, the shell would have looked for a variable called **VG_LVNUM**, which of course does not exist, and we would produce an error message since we no longer have a valid variable assignment statement.

By using **eval** the first scan evaluates **${VG}** to produce the string **rootvg_LVNUM** and at the same time produces the output from the command substitution; the second scan performs the variable assignment. If we had not used **eval**, the shell would make the variable and command substitution, then try to interpret the whole line as a series of characters and so produce an error message since it no longer recognizes the equals sign as an assignment operator.

The **print** command is inserted just to show that the string **rootvg_LVNUM** is printed without extracting the value assigned to it.

The final **eval** command creates an entry in memory containing the characters1 **$rootvg_LVNUM** on the first scan, and on the second scan the value of **rootvg_LVNUM** is extracted.

SHELL VARIABLE $$

In the following example, we make use of a special type of variable maintained by the shell. You cannot assign values in the usual way to automatically maintained variables, but you can reference them.

One such variable is **$$**, which is always equal to the process ID of the current shell process. If you enter **ps**, and then **print $$**, you will see that the process ID of the login shell, **-ksh** (or it may be the PID of */bin/ksh* if you are using multiple windows), is confirmed by **print $$**. Similarly, you can determine the process ID of a process which is running a shell script by including a command such as **print $$** in the script.

For example:

```
$ vi showpid
print The PID of the process running me is: $$
```

and on running the command **ksh showpid** the above message will be displayed with the PID inserted.

Consider the following example, which illustrates that each shell script is indeed run by a separate process. Create the following shell scripts:

```
$ vi one
print In one: $$
date
```

```
ksh two

$ vi two
print In two: $$
```

Now display the process ID of the current login shell, and run **one**:

```
print $$
ksh one
```

and note the different PID numbers.

Here is what happens. When you enter the command **ksh one**, a new shell process is started. This process reads the first **print** command in the shell script, and since **print** is a built-in command, a new process is not started. It then replaces **$$** with its own process ID, which is that of the process running the script, and executes the resultant command.

Next, the shell reads the **date** command, and since it is not a built-in command, **forks** a new process. The new process then **execs** the **date** program. When the **date** process terminates, the shell reads the next command, **ksh two**. The shell again **forks** a new process, and the new process **execs two**.

The operating system recognizes that **two** is a shell script, and overlays the new process with the **ksh** program, and arranges for this **ksh** process to get its input from the script **two**. This shell process then reads the **print** command in **two**, replaces **$$** with its own process ID, and runs the resultant **print** command.

Since this is the last command in **two**, the shell process terminates, and, when it does, the parent shell process running the commands in **one** is re-activated. The parent now finds that it is at the end of file **one**, and terminates. When it does, its parent, the login shell, is re-activated.

You can also try running the script with:

```
. one
```

which will run **one** in the current shell. This will then display the **In one:** PID as the same as that of **-ksh**.

As a further example, suppose that you change **one** to the following:

```
$ vi one
print In one: $$
```

```
date
ksh two
print In one again: $$
print goodbye
```

When **one** is executed, the shell process running one executes **print**, then waits idly while the **date** process runs. When **date** completes, the shell process handles the next command, **two**. Again the shell waits idly for the shell process that is running **two** to complete. When it does, the shell process running the commands in **one** handles the remaining commands in the script.

Tonto Kowalski
Guru (UAE)

Core dumps in AIX 5L

INTRODUCTION

Those of you who have the task of dealing with core dumps may be interested to learn about new functionality in AIX 5L which makes administration of core dumps and problem determination through core dump analysis considerably easier.

CORE DUMP NAMING

The first thing you will notice when dealing with core dumps in AIX 5L is that the naming designation for dumps has changed. Prior to Version 5L, AIX used to name all core files as core. Obviously, if an application dumped more than once, the earlier core dump would be overwritten. This could cause interesting problems with tracing and debugging.

With AIX 5L Version 5.1 a new naming structure gives each core dump a unique name (such as core.17831.18150903). The designation has the following components: core.[process ID].[Day][hour][minute][seconds]. In our example designation shown above, the Process ID was 17831, and the dump was taken on the 18th day of the month, at 15.09 and 3 seconds.

As a result of this simple change, core dumps are no longer overwritten, which is a real bonus for tracing application failures and for debug operations. However, this new 'unique' naming scheme is not the default setting in AIX 5L. The default is still to name the core dump as core. To enable the new naming standard you will need to set the CORE_NAMING variable to yes. In fact, any value except null will be accepted as a yes value.

SNAPCORE

There is another feature found in AIX 5L which helps in problem determination. AIX 5L now uses the **snapcore** command to automate the collection of core dumps and deposit them in a single archive. This is important because all the relevant information relating to a core dump (such as the core file itself, the program, and the libraries) can now be found in a single compressed pax archive in a specified default directory.

The fact that the compressed core dumps are deposited by default in the *tmp/snapcore* directory, unless specified, is a very useful feature indeed. The **snapcore** command will gather all the information relating to a dump and create a new compressed pax archive in the *default/tmp/snapcore* directory. You can of course, use the **-d** flag to change the directory where you want the archive stored. This can be anywhere from disk to tape, or at a remote location.

EXAMPLE

In the examples below, we will run the **snapcore** command to collect all relevant files for our core dump (core.17831.18150903). We will save the dump in a directory that we have created (*/tmp/cdumpdir*). We will then view the contents of the core and run the check_core utility against the core dump file so that we can obtain a list containing the program that caused the core dump and the libraries used by it.

First, we change to the directory where the core dump file is situated:

```
#ls -l
total 43
-rw-r—r—1 root staff 86 Feb 16 13:11 .profile
-rw———1 root staff 432 Feb 16 19:12 .sh_history
-rw-r—r—1 root system 20761 Feb 18 13:09
```

```
core.17831.18150903
```

We can then use the **-d** flag to save the dump to */tmp/cdumpdir* rather than the default */tmp/snapcore* directory:

```
#snapcore -d /tmp/cdumpdir core.17831.18150903
Core file "core.17831.18150903" created by "telnet"
pass1()in progress ....
Calculating space required ..
Total space required is 7985 kbytes ..
Checking for available space ...
Available space is 11361 kbytes
pass1 complete.
pass2()in progress ....
Collecting fileset information ..
Collecting error report of CORE_DUMP errors ..
Creating readme file ..
Creating archive file ...
Compressing archive file ....
pass2 completed.
Snapcore completed successfully.
Archive created in /tmp/cdumpdir.
```

Each execution of the **snapcore** command creates a new archive file. The compressed pax archive file that is created is given the designation *snapcore_<$pid>.pax.Z*. This command uses **$pid** (pid of the **snapcore** command) to create a unique name file and preserve any previously created archives. You can use the **-r** flag to remove the previously created archive file:

```
#ls -l /tmp/cdumpdir
total 4620
-rw-r—r—1 root system 2845109 Feb 18 13:11
snapcore_4680.pax.Z
```

When you need to view the pax archive contents you can use the following command:

```
#uncompress -c snapcore_4680.pax.Z |pax
core.17831.18150903
README
lslpp.out
```

At some point you will probably want to examine the dump, at which point you will need to use the following commands to extract the files from the archive:

```
#uncompress -c snapcore_4680.pax.Z |pax -r
#ls -l
```

```
total 4772
-rw-r—r—1 root system 415 Feb 18 13:21 README
-rw-r—r—1 root system 20761 Feb 18 13:21
core. 17831.18150903
-rw-r—r—1 root system 43682 Feb 18 13:21 lslpp.out
-rw-r—r—1 root system 2845109 Feb 18 13:11
snapcore_4680.pax.Z
```

The utility used by the **snapcore** command to gather relevant information about the core dump is called check_core. The check_core utility is a C program located in the */usr/lib/ras* directory. However, to use check_core you need to make sure you have the bos.rte.serv_aid fileset installed. When you run the check_core utility against the core dump file you will receive a list containing the program that caused the core dump and the libraries used by it:

```
#/usr/lib/ras/check_core core.17831.18150903
/usr/lib/libc.a
/usr/lib/libcrypt.a./usr/lib/libbsd.a
/usr/lib/libbind.a
/usr/lib/libi18n.a
/usr/lib/libiconv.a
/usr/lib/libcur.a
/usr/lib/libauthm.a
/usr/lib/libodm.a
/usr/lib/libcfg.a
/usr/lib/nls/loc/en_US
telnet
```

As you can see, check_core provides the name of the program that caused the dump. Also you will know it from the core dump error log.

Removing core archives

At some point you may wish to remove old core archives. To remove the previously created core archive in the */tmp/snapcore* directory and create a new one, enter the following:

```
snapcore -r<core file name> <program name>
```

To create the *core* file archive in an alternate directory, enter the following:

```
snapcore -d<dir name> <core file name> <program name>
```

The pax file is created in *<dirname>/tmp/snapcore* directory.

To clean the */tmp/snapcore* directory, enter the following:

```
snapcore -r
```

CONCLUSIONS

In our shop we have found the new core dump naming convention and the **snapcore** command to be a significant bonus in debugging and tracing application failures. The pax archive contains all the relevant information that is required to analyse a core dump and it is placed in a standard directory. It is one of the many new and useful features of AIX 5L.

Systems Programmer (UK)

Understanding the head and tail commands

HEAD COMMAND BASICS

Suppose you had several files in a directory and you wanted to know their purposes. If you could display just the first few lines of each file, perhaps it would give you enough data to determine the files' intent. With the **head** command you can do this.

Or let's say you have 30 files whose contents you know, but you need to scan through them and see a certain field 255 bytes into the file. The **head** command can help you.

As a further example, suppose you had 180 source program files, each of which contains a 12-line prologue. Let's say you need to concatenate the prologues of the 180 files into a single file for the purpose of running further filters against it. Using the **head** command will make this easier.

The **head** command displays the first few lines of a text file depending on a line count you specify. You can specify a byte count instead, and the **head** command will display that number of bytes beginning with the first byte of the file.

If you don't include an input count on the command line, the **head**

command will display the first ten lines of the file you specify on the command line.

TAIL COMMAND BASICS

Now let's look at the **tail** command. This command displays the LAST group of lines or bytes of the specified file or files.

Using the **head** command examples as models, let's address some similar uses for the **tail** command.

Let's say you have several log files whose data consists of entries written chronologically – the latest items at the bottom of the file. If you wanted to display the last few lines of each log, the **tail** command will do this.

Or maybe you have a group of files with key data within the last 100 bytes in each file. The **tail** command will show this to you.

Another example would be if you wanted to concatenate the coded portions of your 180 source files from the **head** command example above into a single file or filter command. The **tail** command displays the last few lines or bytes of a text file, based on the count you specify on the command line.

As the head command displays the first ten lines as a default, the **tail** command will display the last ten lines of a file if no count is specified.

The **tail** command offers two further benefits. It can continuously display the last entries of a log file that is being updated by an independent process. It can also display the data in a file in reverse order, such as if you wanted to display a chronological event log with the most recent entry first.

The basic syntax of the **head** and of the **tail** commands is shown below:

```
head flags filespec
tail flags filespec
```

where:

- *flags* is an optional flag or flags used to enhance the head or tail operation.

- *filespec* is the file or files on which the **head** or **tail** operation is to perform.

FLAGS FOR THE HEAD COMMAND

The following flags help to enrich the usefulness of the head command.

- -c number – specifies the number of bytes to display.

  Use the **c** flag to tell the **head** command how many bytes from the beginning of the file you would like to display. For example, **head -c 100 headsmp1.fil** would display the first 100 bytes of the sample file, whatever those bytes might be, as long as headsmp1.fil is a text file.

- -n number – specifies the number of lines from the file's beginning to display.

  Use the **n** flag to tell the **head** command how many lines from the beginning of the file you would like to display. For example, **head -n 20 headsmp2.fil** would display the first 20 lines of the sample file.

You can specify either the **c** flag or the **n** flag on the **head** command. The *number* parameter in the flags of the **head** command must be a positive integer.

FLAGS FOR THE TAIL COMMAND

The following flags help to enrich the usefulness of the tail command.

- -c number – specifies byte count from the end of the file to display.

  The **c** flag tells the **tail** command, Start at this byte count from the end of the file. For example, if you were to enter **tail -c 24 tailsmp1.fil** and if the sample file were 1024 bytes in size, the **tail** command would display the last 24 bytes of the file.

- -f – specifies that the tail command is not to terminate after the display.

  The **f** flag tells the **tail** command to continue to display data until stopped. Suppose you had a program continuously writing to a

log. If you entered **tail <process>.log**, where <*process*> is the program writing to the log file, **tail** would at first display the last ten lines of the log file. As the program continues to write to the log, the **tail** command would continue to append the new lines to the end of its current display either until stopped with the **Ctrl+C** command (or otherwise stopped,) or until the program stopped writing to the log. This would enable you to monitor a log file without editing the file.

- -n number – specifies a line count from the end of the file to display.

  The **n** flag tells the **tail** command, Start this many lines from the end of the file and display from that point. For example, if tailsmp2.fil had 500 lines, **tail -n 100 tailsmp2.fil** would start at line 401 and display the last 100 lines of the file. This could be useful if you wanted to scan only the data in the last few lines of your files.

- -r – specifies that the **tail** command is to display from the file's end in reverse order.

  The **r** flag tells the **tail** command to display the data beginning with the last line first, followed by the second to the last, etc, until the line number specified using the **n** flag. This could be used to display a log file in such a manner, if it were to be more meaningful to read the last line first. If no **n** flag is specified, the **tail** command will display the entire file in reverse order.

The *number* parameter in the flags of the tail command can be a positive or negative integer. If no sign, or the minus (-) sign, is specified, the number of lines is displayed from the bottom of the file. If the plus sign (+) is used, the tail command will display lines or bytes from the beginning of the file.

For the examples above, entering **tail -c +1000 tailsmp1.fil** would bypass the first 1000 bytes of the 1024 byte file and display the last 24 bytes. Entering **tail -n +400 tailsmp2.fil** would bypass the first 400 lines and display the last 100 lines of the 500 line file.

It is not at all uncommon to pipe the output of either the head or the tail command into the *pg* or other filters to enable you to see the results one page at a time.

SOME EXERCISES

Here are some exercises to test your knowledge of the head and tail commands.

Exercise 1 – head command exercises

To set up the exercise:

1 Create three sample files using the code at the end of this article as a model.

2 Name the files sample1.src, sample2.src, and sample3.src.

3 Edit sample2.src and sample3.src and change the following:

- Increment the Module ID in line 1 to HT002 and HT003.
- Increment the Module Name in line 4 and the Source file number in line 6.
- Change the Date, Programmer initials, and Reason for Flag $A2 in line 21 to be different from sample1.src.
- Place the three sample files in a test directory.

Step 1 – enter head sample1.src.

Observe the following results:

```
// START OF SPECIFICATIONS - Module ID: HT00 1
// ***************************************************************
//
// Module Name:   sample1.src
//
// Description:   Source file number 1 for head tail exercise.
//
//---------------------------------------------------------------
//      ABC Company
//---------------------------------------------------------------
```

You have just used the **head** command to display the first ten lines of the first sample file. This can help you determine the intent of a file if you or another developer were unfamiliar with it.

Step 2 – enter head -c 45 *.src.

Observe the following results:

```
==> sample1.src <==
// START OF SPECIFICATIONS - Module ID: HT001
==> sample2.src <==
// START OF SPECIFICATIONS - Module ID: HT002
==> sample3.src <==
// START OF SPECIFICATIONS - Module ID: HT003
```

You have just used the **head** command with the **c** flag to display only the first 45 bytes of each of the three sample files. This could help you if you knew in advance that the data that you were seeking was within the first 45 bytes of each file; in this case, the Module ID.

Step 3 – enter head -n 25 *.src | grep A2.

Observe results similar to the following, depending on how you changed the samples:

```
// $A2   05/03/2001   MKS   Changed data_x variable
// $A2   05/05/2001   RTH   Changed data_j array
// $A2   05/07/2001   KLR   Changed data_f constant
```

You have just used the **head** command with the flag to isolate only the 25 line prologue of all your .src files as input to the **grep** command. This would display the change flags only in the prolog, and not in the body of the code. This can help you if you wanted to ignore the change flags in the actual code, and see only the change flag in the prolog.

Exercise 2 – tail command exercises

There are two log files that will be utilized during this exercise. Sulog is a file that records whenever a user uses the **su** (switch user) command. In a typical system, */var/adm/sulog* contains many entries. If your copy is sparse, you may wish to issue several instances of **su** in order to log enough entries to perform the exercise. Alternatively, you may use another system log for the exercise. You must have system authority to access sulog.

Smit.log keeps track of activity during use of the System Management Interface Tool (smit). This file may typically be found in the root directory for root users or in the home directory of regular users. If there is no smit.log in your directory, you will need to run smit (smitty for command line use) and run a few operations to develop a log.

Once you have sulog and smit log available, you are ready to begin the exercises.

Step 1 – as an administrative user, enter tail -n 8 /var/adm/sulog.

Observe results similar to the following:

```
SU 10/13 09:53 - lft0 wjsmith-root
SU 10/13 10:05 - pts/0 wjsmith-root
SU 10/13 10:06 - pts/0 wjsmith-root
SU 10/13 10:06 - pts/0 wjsmith-root
SU 10/13 10:07 + pts/0 wjsmith-root
SU 10/13 10:07 - pts/0 wjsmith-root
SU 10/18 15:44 + pts/0 baines-root
SU 10/23 15:22 + pts/0 baines-root
```

You have just used the **tail** command with the **n** flag to display the last eight entries of the log file that tracks the use of the **su** command. This can be useful if you want to view only the most recent activity in a log file.

Step 2 – as an administrative user, enter tail -r -n 5 /var/adm/sulog.

Observe results similar to the following:

```
SU 10/23 15:22 + pts/0 baines-root
SU 10/18 15:44 + pts/0 baines-root
SU 10/13 10:07 - pts/0 wjsmith-root
SU 10/13 10:07 + pts/0 wjsmith-root
SU 10/13 10:06 - pts/0 wjsmith-root
```

You have just used the **tail** command with the **r** flag to view the last five lines of the **su** log file displayed in reverse chronologic order. Sometimes viewing a list beginning with the last item first is more useful.

Step 3 – cd to your root or home directory and enter tail -c 255 smit.log.

Observe that the command displays the last 255 bytes of the smit.log.

You have just used the **tail** command with the **c** flag to view a specific byte count from the end of a file.

Exerxise 3 – using the tail command to monitor a process.

Part two requires two command windows to be used – one to run commands that will write data to a log file, and one to run the tail command to view the log files.

Step 1 – from command window 1, enter tail -f /var/adm/sulog.

The **tail** command will display the last ten lines of the log, and trap the

cursor. From command window 2, enter a few **su** commands. Observe that for each **su** command you complete in window 2, a new entry will be appended to the results in window 1. Press **Ctrl+C** in window 1 to halt the **tail** command and to return the cursor.

You have just used the **tail** command with the **f** flag to dynamically monitor the update of a log by another process. This can be useful if you need to keep track of an independent, remote process.

Step 2 – cd to your root or home directory and from command window 1, enter tail -f -n 20 smit.log.

The **tail** command will display the last twenty lines of the log and trap the cursor. From command window 2, enter **smit** (smitty for command line). For each **smit** command that completes, you should see new entries added to the end of the log displayed by the tail command in window 1. Press **Ctrl+C** to halt the tail command.

You have used the **tail** command to monitor another external process.

Exercise source code

```
// START OF SPECIFICATIONS - Module ID: HT001
// ********************************************************************
//
// Module Name:   sample1.src
//
// Description:   Source file number 1 for head tail exercise.
//
//--------------------------------------------------------------
//      ABC Company
//--------------------------------------------------------------
//
// Author:  William J. Baines
//
// Function:   Helps you to learn the AIX head and tail commands.
//
// Change Activity:
//
// FLAG  DATE         PROG  REASON
// ──  ─────   ──  ─────-──────────-
// $A1   04/02/2001   WJB   Updated matrix table
// $A2   05/03/2001   MKS   Changed data_x variable
// $A3   02/01/2002   WJB   Deleted duplicate Y string
//
// ********************************************************************
// END OF SPECIFICATIONS
```

```
//
//    — Start of Code —-

                  .
                  .
                  .

//    — End of Code —-
//
```

David Chakmakian
Programmer (USA)

Removing users from a system

INTRODUCTION

In the current economic climate, the need to remove users from systems is unfortunately becoming a common task. This is something that has to be given top priority in a system administrator's daily workload because the penalty for slow action can be considerable.

The computer press would have us believe that the major risk to systems comes from external attack by malicious hackers. In reality, however, most attacks on systems originate from within the network. It is currently thought that between 65% and 90% of network attacks come from within an organization rather than originating from outside. These numbers are difficult to quantify because there is still a strong reluctance for companies to admit that their networks have been compromised, and partly because skilled network breaches may be difficult to identify and may even not be properly recognized until long after the event. It is, therefore, very difficult to deduce the origins of different attacks.

The principal worry for network security staff and systems administrators is that disaffected staff, or former workers, may have deliberately left security breaches or even time-bombed an installation before quitting. Another important area of exposure is disgruntled users who retain their user ID after they have left. This is why the removal of user access

privileges and associated data must be undertaken as soon as possible after the personnel have left.

USER REMOVAL TASKS

When a user is to be removed from a system permanently, the following tasks need to be performed.

Remove user information

All user information must be removed. To remove a user you can use the **smit rmuser** command. For example:

```
User NAME                                    [Jack]
Remove AUTHENTICATION information?             yes
```

However, before this is attempted it is essential to check the user's files. This can be achieved with the **find** command (before the user is removed).

find

The **find** command recursively searches the directory tree for each specified *Path* parameter. It uses a Boolean search to find files. However, remember that when the **find** command is recursively descending directory structures, it will not search directories that are symbolically linked to the current hierarchy.

```
find -user jack
```

It is important to note that if the user's files and the user's account are deleted before reassigning the user's files then the files will essentially become 'unowned', and can become a security risk.

All the user's files on the system need to be removed or reassigned. So it is a simple matter of using the **find** command to generate a list of all files owned by the user. If the files are useful, they can be reassigned to other users using the **chown** command or else deleted.

However, it is essential that you remove information in other subsystems before removing a user, because the **cron** and **at** utilities both allow users to request programs to be run at a future date. Use the **crontab** command to remove a user's **cron** jobs. You can examine a user's **at**

jobs with the **atq** command, and then remove the jobs with the **atrm** command.

If you need to remove an entire group the **smit rmgroup lsgroup -a** *Attributes Group* procedure removes a group and all of its attributes from your network. However, this procedure does not remove all of the users in the group from the system. Furthermore, if the group to be removed is the primary group for any user, you must reassign that user to another primary group before removing the user's original primary group.

at jobs

It is absolutely crucial that you remove any **at** jobs that the user has scheduled. This is because a user can use the **at** command to schedule potentially damaging programs to run long after they are removed from the system. This can be achieved by using the **cronadm** or the **atq** command.

cronadm

The **cronadm** command (found in */usr/bin/cronadm*) is used by a root user to list or remove all users' **crontab** or **at** jobs. The jobs are listed and removed by the *UserName* parameter. One or more *UserNames* can be specified. The **at** jobs are listed by *UserName* and can be removed either by the *UserName* parameter or by the *JobName* parameter.

When using the **cronadm at** command, certain flags will be useful. **-l** lists the **at** jobs for the user specified by the *UserName* parameter. For example, if you want to list all **at** jobs currently queued for the user jack, simply enter:

```
cronadm at -l jack
```

The **-r** flag removes the **at** job specified by either the *UserName* or *JobName* parameter.

```
cronadm at -r jack
```

atq

The atq command (found in /usr/bin/atq) displays the queue of jobs

waiting to be run at a later date, sorted in the order the jobs will be run. These jobs were created with the **at** command. If the user is root and *User* name is specified, the **atq** command displays only jobs belonging to that user. The **-c** flag sorts the queue by the time that the **at** command was issued, while the **-n** flag displays only the number of jobs currently in the queue.

Example

In order to look at the queue created by the **at** command, enter:

```
atq
```

If there are jobs in the queue, a message similar to the following appears:

```
root.635623200.a          Wed     Feb 20  13:00:00 2002
root.635670000.a          Fri     Feb 22  15:00:00 2002
```

atrm

An alternative command for root users is **atrm** (found in */usr/bin/atrm*). This removes jobs that were created with the **at** command, but have not executed. If one or more job numbers is specified, the **atrm** command attempts to remove only those jobs. If one or more user names are specified, all jobs belonging to those users are removed. This form of invoking the **atrm** command is useful only if you have root user authority.

Example

To remove job number root.62169200.a from the **at** command queue, simply enter:

```
atrm root.621619200.a
```

Temporary removal

If the user is being removed only temporarily, consider just removing the ability of the user ID to log in to the system. You can also use the */var/adm/cron/at.allow* and */var/adm/cron/at.deny* files, which control which users can use the **at** command. A root user can create, edit, or delete these files. Entries in these files are user login names with one name to a line.

If the *at.allow* file exists, only users whose login names appear in it can use the **at** command. A system administrator can explicitly stop a user from using the **at** command by listing the user's login name in the *at.deny* file.

CONCLUSIONS

It is absolutely crucial that security staff or systems administrators remove the access privileges and data from employees who have left as soon as possible. The techniques shown above are the principal methods you should consider.

Systems Programmer (UK)

Variables and the environment

THE ENVIRONMENT OF A PROCESS

Like any other programming language, the shell provides capabilities for creating variables, assigning them values, and accessing those values. Shell variables may be assigned a string or any sequence of characters.

Every process has an *environment*, which you can think of as a list of the variables that can be passed to a child process. When a shell process starts running, it makes a copy of all the variables in its environment, and you can reference and modify the local copy of these variables in the same way that you use other shell variables. Note that when you change the value of one of these variables, you affect the local copy only.

The environment with which a new process is started is the same as the parent process's environment. When the new shell process is started, it inherits its parent's environment and makes a local copy of all the variables in the environment, and we shall later create scripts to observe this.

When a child process reads a command and itself starts another process, the new process will also inherit the parent shell's environment. If the local value of one of the variables in the parent shell's environment has been altered, the environment that the new process inherits will contain this new value, not the original value.

You can display your login shell's environment by using the **env** command. Do this, and note that the output looks similar to that of **set**, except that the difference between **set** and **env** is that **set** displays all the variables that exist in the current process (including those assigned a null string and those defined during your login session and which will be lost when you log off), while **env** displays only those that are included in the process's environment, and so are available in sub-shells.

How is the environment created?

Although we tend to think of environments in the context of users, the environment of every process is created by:

- Inheriting variables from */etc/environment.*
- Inheriting variables created by other processes, or by modifying existing inherited variables; for users in particular this manifests itself in the execution of the */etc/profile*, *.profile*, and other similar files.
- Inheriting other system variables.

THE /ETC/ENVIRONMENT FILE

The */etc/environment* file is used to specify the basic environment for each and every process, not just those executed by users. When a new process begins, this file is read, and any variables declared are placed in an array, which becomes the process's initial environment – this array is a local copy of the set of variables. Unlike the profile files, the environment file is not a shell script and does not accept data in any format other than *name=string*. The file contains variable assignments such as those for **TZ**, **LANG**, default **PATH**, **NLSPATH**, etc.

A running process's environment is not changed by altering the

contents of the */etc/environment* file and any processes started prior to any changes you make will still access the local copies they made. Only when the process is restarted do the changes take effect, which for some processes will be a necessity. For example, if you change the **TZ** (time zone) variable, you will have to restart **cron**. The contents of */etc/environment* can only be changed by **root**, or members of the **system** group.

USER LOGINS

User logins are perhaps the best example of how environments are modified by inheriting variables created by other processes.

When a user logs in, the login process first reads the */etc/environment* file, then */etc/profile* is executed, followed by *.profile*. If the user is in an Xwindows environment then a number of other similar scripts will be executed. If you have created a *.kshrc* file, then this also may contribute to your environment after the *.profile* is executed. Each stage adds new variables, or modifies existing ones, until the user's full working environment is established.

The */etc/profile* file is a system-wide file, which is executed by all users at login. The file, amongst other things, sets the values of variables that are used by every user, such as **TMOUT**, **MAIL**, and **MAILMSG**. It is possible, provided you are **root**, to modify this file to define values for other variables, or to change existing ones. If you wish any variables to be available in sub-shells, then they must be exported. Entries in the file should be restricted to those that relate to all users.

/etc/profile is normally executed only as part of the login shell, and, should the file be modified later, then, to access any changes without logging off and logging back in again, you must execute the file using the **dot** command. You can run any shell script this way, even if it is not currently in execution mode (it must not be an ordinary text file, but must be capable of being executed, however). When you use the **dot** command to run a shell script, the commands in the shell script are run by the current shell process, *not* by a sub-shell.

You can override the values set in */etc/profile* by changing them in your user *.profile*, which is executed after */etc/profile*. If you make

modifications to any variables which may be contained in the file, for example your **PS1** variable, then similarly you must use the **dot** command to make the changes available in your current shell.

SYSTEM VARIABLES

When the operating system boots up, it creates a number of system variables, nearly always notated in capital letters, and assigns default values to them. These variables will always be available, and, no matter what process or sub-shell you are in, you can always access their values. The results of any modifications you make to these values depend entirely on whether or not the variable has been exported.

Non-exported system variables

The best way to illustrate what happens when you modify non-exported system variables is to use an example. On entering the **set** command to list the variables available for the login shell, you will notice a system variable named **MAILCHECK**, which has a default value of **600**. If you enter **env** you will notice that **MAILCHECK** is not present in the list and so is not part of your environment.

If you create a sub-shell by entering the **ksh** command, and again enter **set**, you will see **MAILCHECK** is still in the list, thus signifying that it is available to be accessed by the current shell/process. Similarly, **env** will show that it is still not present.

Next, change the value of **MAILCHECK** with the command **MAILCHECK=100**. On using the **set** command again, you will note that its value has been changed. Now create a further sub-shell with **ksh** again, and then enter **set**. You will now see that **MAILCHECK** still has its original value of **600**.

This indicates that non-exported system variables passed to the second sub-shell will retain the values of the original login shell and not the modified values of the first sub-shell. The reason for this is that alterations to the first sub-shell are made only on copies of the originals, which, because they are not exported, will not be passed on to dependent processes.

Exported system variables

Once a system variable has been exported, any modifications that you make will be passed on to dependent processes. This can be illustrated by continuing with the above example. You should at this stage be in the second sub-shell. Press **Ctrl+D** to cancel this sub-shell and return to the first sub-shell, and enter the command **export MAILCHECK**.

When you now enter the **env** command, you will note that **MAILCHECK** appears on the list with its new value of **100**. Now create a second sub-shell again with the **ksh** command, and on entering **set**, you will see that **MAILCHECK** is currently available for this sub-shell, and has the new value of **100**. Any further sub-shells you create will now have this new value passed on to them. On entering **env** each time, you will see that it is now part of the environment.

If you now exit from all these sub-shells, and return to the original login shell, on entering the **set** command you will note that **MAILCHECK** has retained its original value of **600**. The **env** command will also not display **MAILCHECK**. This illustrates that modifying the environment of a process *does* affect the descendent processes, but *does not* affect the parent process.

USING SYSTEM VARIABLES IN SCRIPTS

We will now run sample shell scripts to demonstrate that:

- System variables that are part of your login shell's environment are also part of any shell script's environment.
- If you modify system variables that are part of the shell script's environment, you will affect the local copy only. The modified variable will affect descendent processes, but when you exit the shell script the login shell's original environment will not be altered.

Create the following shell script, **showenv**:

```
print "The login shell's value of TERM is:" $TERM
print
# modify a variable from the environment
TERM=hi
print In showenv TERM is now: $TERM
```

```
print
# Show that the script's environment is changed
env | grep TERM
```

If you now execute the script with **ksh showenv**, note the values of **TERM**. If you then display the value of **TERM** in the login shell's environment again with the **print $TERM** command, you will note that **TERM** retains its original value.

If we modify **showenv** so that it invokes another shell script, we can show what happens when we again change the value of **TERM**. Modify **showenv** and replace the last line with **ksh myenv**. Now create a new shell script, called **myenv**, containing the following lines:

```
print In myenv TERM is: $TERM
# set TERM to new value
print TERM=hello
print Again in myenv, TERM is now: $TERM
```

When you execute **showenv**, note that **TERM** changes as each shell script modifies its value. If you finally enter the command **print $TERM**, you will note that the original value is unchanged.

Running **showenv** thus demonstrates that the shell process started by this script receives an environment containing the original value of **TERM**. When this value is modified, the modified value is passed to the new environment started by **myenv**. When **myenv** and **showenv** finish executing, the original value of **TERM** is unchanged.

USING OTHER VARIABLES IN SCRIPTS

Unlike system variables, any variables that you create will not be accessible to descendent processes unless they are explicitly exported. As soon as you create a variable, it is added to the list of variables available for that process/sub-shell; the **set** command will show that it has been added to the list.

What happens with this type of variable is best shown with an exciting(!) example. Note that with all the scripts in this article we have not included the standard first line, **#!/bin/ksh**, since we are only creating very simple scripts. In your login shell create a variable **var1**, and assign it the value **xxx**. Use **set** to show that it is available. Create a sub-shell with **ksh**, and, if you again use **set**, you will see that its name

does not appear on the list.

Now create the same variable **var1**, but assign it the value **yyy**. **set** will show that it exists. Now exit from the sub-shell, and on entering **set** again you will see that **var1** retains its original value. This is because the variable created in the sub-shell is only a local copy, even though it has the same name as the variable in the login shell.

Now **export var1**, and, if you are still awake, you will see that, on executing **env**, the variable **var1** is now included in the environment. Create a new sub-shell, enter both the **set** and **env** commands, and if the excitement has not got the better of you, you will again see that in this mind-bogglingly boring example the variable is included in both lists!

You can also create the following shell scripts to illustrate this further:

```
$ vi one
var1=xxx
print In one, var1 is: $var1
print
ksh two
print
print Back in one, var1 is: $var1

$ vi two
print In two, we inherited var1: $var1
print
ksh three
print
export var1=yyy
print Back in two, var1 is now: $var1
print
ksh three

vi three
print In three, we inherited var1: $var1
```

When you execute **ksh one**, you will see the following output:

```
In one, var1 is: xxx
In two, we inherited var1:
In three, we inherited var1:
Back in two, var1 is now: yyy
In three, we inherited var1: yyy
Back in one, var1 is: xxx
```

The second and third lines of the output show **var1** to be null since we are using only local copies at this stage and these have not yet been set.

In **two** you will see that we have exported **var1**. Note that it does not matter when you **export** a variable as long as you do so before starting any processes that reference it. You can **export** the variable before you assign it a value, after you assign it a value, or in the Korn shell, at the same time you assign it a value, as we have done above. Once it has been exported, its current value will always be available to descendent processes.

The variable **var1** can be made available in your current shell, in this case your login shell, by running the script with:

```
. ./one
```

By running scripts in this way it is possible to modify any variables that currently exist in your login shell. You can similarly access the variables from any script by including the line:

```
. other_script_pathname
```

in the script you are running. Many shell programmers often create a file just containing commands which assign values to variables since this then allows the file to be executed by different, and usually related, scripts. In this way you need only to maintain a single copy of the variable assignment, thus making it easier to alter without having to edit a number of scripts.

SUMMARY

All the above examples show that:

- Each time a shell script is run, it is run by a separate shell process.
- Every shell script inherits an environment from its parent, and makes local copies of each of the variables in the environment. Modifications to the copies are passed on to descendent processes.
- When a variable is exported within a process, the environment is updated with the current value of the latest local copy, and the value will be available for descendent processes.

- A process cannot add variables to the parent process's environment, and cannot modify the values of any variables in the parent process. Processes can only affect their descendents.

DEBUGGING USING THE -X OPTION TO KSH

The **-v** option to **ksh** causes **ksh** to print the commands in a shell script before they are executed. This particular option merely prints the command, and then its output on the next line, but it can be difficult to distinguish the command itself from its output. In simple shell scripts this is not usually a problem, but in complex scripts with lots of functions, or where there are nested scripts, such as the example above, it can be a problem.

When you are debugging complex scripts, it is often more useful to use the **-x** option to **ksh**. For example, if you entered the command:

```
ksh -x one
```

to execute the file you previously created, you will notice that all of the command lines in **one** are printed and preceded by a + before they are executed and it is now much easier to distinguish the command itself from its output.

Since the commands in **two** and **three** are executed in sub-shells, their commands are not displayed since **ksh -x** only operates on the sub-shell in which **one** is running. You could, if you wished, call either of these scripts from **one** with the **-x** option, for example **ksh -x two**, and this would allow you to concentrate on debugging the scripts one at a time (not that this would be a particularly onerous task for these scripts!).

If you are using the shell interactively, you can set this option by entering the command:

```
set -x
```

After entering this, all succeeding commands you enter in the current process will be displayed and preceded by a + before executing. To turn the option off, enter:

```
set +x
```

You can also add the **set -x** command at the beginning of a shell script,

and **set +x** at the end, although the latter is not necessary at the end of a file, or you can enter these commands at any point during a shell script to debug only a limited number of lines of code. Try this on the **two** file.

You should be aware that if you have command line arguments for the script you intend to run with **ksh -x**, then, unless you specify the full pathname of the script, you will get an error message that the command cannot be found. Rather annoying, but what else do you expect from the Unix world!

You should be further aware that if you use **set -x** to debug scripts that contain functions, then the individual commands contained within the functions will not be displayed unless you use the same **set** commands within the function itself.

Tonto Kowalski
Guru (UAE)

Advanced features of EMC PowerPath software

EMC PowerPath is a software layer that enables efficient management of I/O for AIX-based server connected to Symmetrix storage systems. This article provides a description of more advanced usage of the product.

USAGE OF A POWERPATH VOLUME AS AN AIX BOOT DEVICE

The PowerPath hdiskpower devices that contain the AIX boot image must provide I/O load balancing and failover protection for the host's boot device. The Symmetrix volume used for this purpose must fulfil the following conditions:

- The boot device has to be connected using a SCSI adapter
- The boot device should be visible only to a single host
- The host's boot list must contain all the hdisks that comprise the boot device.

The following procedure should be used to set up the PowerPath boot device for a new PowerPath installation:

1 Reduce the number of physical paths to the intended boot device to 1.

2 Install AIX on a Symmetrix volume connected using a SCSI adapter.

3 Install the latest EMC Symmetrix drivers.

4 Reboot the host.

5 Install PowerPath software. Do not run the **powermt config** command!

6 Create a boot image using the **bosboot** command:

```
bosboot -a
```

7 Reconnect all physical paths to the Symmetrix.

8 Make sure that an hdisk has been configured for each path that was reconnected.

9 Execute the command:

```
powermt config
```

10 Use the **pprootdev** command to set up multipathing on the root device:

```
pprootdev on
```

11 Use the **bootlist** command to add all the alternate path hdisk devices to the boot list.

12 Reboot the host.

The following procedure should be used to set up a PowerPath boot device on a Symmetrix for an existing PowerPath installation. This procedure describes the process involved in the migration of the contents of the physical volumes that are contained in the volume group **rootvg** from internal disks to the Symmetrix volumes:

1 Make sure that the following patches have been installed on your host:

– IX62417, IX68140, IX74905, IX74041, IX66626 for hosts running AIX 4.2

– IX73591 for hosts running AIX 4.3.

2 Execute the command **powermt display dev=all** in order to determine the hdisk that corresponds to the hdiskpower device that is intended to be used as the boot disk.

3 Add the corresponding physical disk to the root volume group. If your disk is hdisk1 execute the command:

```
extendvg -f rootvg hdisk1
```

4 Remove the PowerPath software from the system by executing the command:

```
installp -u EMCpower
```

5 Check the number of physical partitions (PPs) available on the EMC Symmetrix volume to be used. The boot disk must have sufficient room to contain the number of PPs used by the volume group **rootvg**.

Execute the following command:

```
lsvg -p rootvg
```

You should see output similar to the following:

```
rootvg:
PV_name   PV STATE  TOTAL PPs FREE PPs  FREE DISTRIBUTION
hdisk0    active   348       48        48..00..00..00..00
hdisk1    active   495       495       99..99..99..99..99
```

You can see that the current boot disk hdisk0 is using 348-48=300 PPs on hdisk0 and that the target hdisk1 contains 495 free PPs.

6 Execute the following command to identify the logical volume that contains the boot image (hd5 is the default):

```
lsvg -l rootvg|grep boot
```

You should see output similar to the following:

```
rootvg:
LV NAME   TYPE LPs  PPs  PVs  LV STATE  MOUNT POINT
hd5  boot 1    1    1    colsed/syncd  N/A
```

7 Verify that the logical volume containing the boot image is located on a disk that you are going to migrate:

```
lsvg -l hd5
```

You should see output similar to the following:

```
hd5: N/A
PV   COPIES    IN BAND   DISTRIBUTION
hdisk0    001:000:000   100% 001:000:000:000:000
```

8 Migrate the logical volume containing the boot image:

```
migratepv -l hd5 hdisk0 hdisk1
```

9 Clear the boot record on the original disk:

```
mkboot -c -d /dev/hdisk0
```

10 Update the boot image on the new destination disk:

```
bosboot -a -d  /dev/hdisk1
```

11 Migrate the content of the source physical volume to the destination disk:

```
migratepv hdisk0 hdisk1
```

12 Remove the original disk drive from the rootvg:

```
reducevg rootvg  hdisk0
```

13 Update the bootlist to use the new boot device. Include all hdisk devices that are contained in the hdiskpower device that will be used as a boot device:

```
bootlist -m normal hdisk1 hdisk9
```

14 Install the PowerPath software.

15 Reboot the host.

16 Set up multipathing for the root device:

```
pprootdev on
```

17 Reboot the host.

DISABLING THE USAGE OF A POWERPATH VOLUME AS AN AIX BOOT DEVICE

The following procedure should be followed to disable a PowerPath

volume from being used as an AIX boot device:

1 Disable multipathing for the root device:

```
pprootdev off
```

2 Reduce the number of physical Symmetrix paths to the boot disk to 1.

3 Reboot the host.

4 Create an updated boot image:

```
bosboot -a
```

5 Reboot the host.

INTEGRATION OF POWERPATH WITH HACMP CLUSTER SOFTWARE

The following procedures describe only the basic steps needed in order to install and configure PowerPath and HACMP. Review Chapter 9, *IBM CLUSTER in Symmetrix High Availability Environment Product Guide* for a detailed description of usage of HACMP with the Symmetrix.

We will consider three different scenarios: installing PowerPath and HACMP on a new host; integrating HACMP in an environment in which PowerPath devices are already present; integrating PowerPath into an existing HACMP cluster.

INSTALLING POWERPATH AND HACMP ON NEW HOSTS

To install PowerPath and HACMP on new hosts do the following:

1 Prepare cluster hardware; verify the functionality of all network and disk connections among the hosts and the Symmetrix.

2 Select a host to perform the initial definition of volume groups controlled by HACMP. Perform the following steps on the selected host:

   a Install PowerPath software.

   b Execute the command **powermt display dev=all**. Identify a PVID for each PowerPath controlled device.

   c Execute the command **lspv** to identify volume groups, followed

by **lsvg –p <volume_group>** to identify volume groups containing the PowerPath controlled devices.

d Install HACMP software. Configure HACMP to use volume groups identified in *step c*. If PowerPath is used in a concurrent resource group environment the command **/usr/lpp/symmetrix/bin/symcurrent** has to be executed in order to fix the file **/usr/sbin/cluster/diag/clconraid.dat** to use PowerPath devices.

e Stop all applications that use the volume groups identified in *step c*. Unmount all file systems that reside on these volume groups. Execute the command **varyoffvg <volume_group>** for each volume group to deactivate it.

3 For each of the rest of the hosts participating in the cluster, perform the following steps:

a Install PowerPath software.

b Execute the command **powermt display dev=all**. Identify PVIDs for each PowerPath controlled device. Any disk that shows no PVID or displays a PVID different from the one seen on a different host has to be removed using the **rmdev** command.

c Execute the script **emc_cfgmgr.sh** followed by the command **powermt config** to configure the PowerPath devices.

d Execute the **importvg** command using PowerPath devices and volume groups defined on the first configured host.

e Execute the **chvg** command for each volume group, changing the auto activation status of the group from the default **yes** to **no**.

f Install HACMP software. Configure HACMP to use imported volume groups.

4 Start cluster services by executing the **clstart** command on all hosts. The volume groups and the underlying PowerPath devices will start to operate under the control of the HACMP software.

INTEGRATING HACMP INTO THE EXISTING POWERPATH ENVIRONMENT

To integrate HACMP into the existing PowerPath environment do the following:

1 Prepare cluster hardware; verify the functionality of all network and disk connections among the hosts and the Symmetrix.

2 For each of the hosts participating in the cluster perform the following steps:

   a Execute the command **powermt display dev=all**. Identify a PVID for each PowerPath controlled device. Any disk that shows no PVIDs or displays a PVID different from the one seen on a different host has to be removed using the **rmdev** command.

   b Install HACMP software. Configure HACMP to use volume groups identified in the previous step. If PowerPath is used in a concurrent resource group environment, the command **/usr/lpp/symmetrix/bin/symcurrent** has to be executed in order to fix the file **/usr/sbin/cluster/diag/clconraid.dat** to use PowerPath devices.

   c Stop all applications using the identified volume groups. Unmount all file systems that are residing on these volume groups. Execute the command **varyoffvg <volume_group>** for each volume group to deactivate it.

3 Start cluster services by executing the **clstart** command on all hosts. The volume groups and the underlying PowerPath devices will start to operate under the control of the HACMP software.

INTEGRATING POWERPATH INTO THE EXISTING HACMP ENVIRONMENT

Integrating PowerPath into the existing HACMP environment is the most complex procedure, and is supported only for standby clusters with cascading resource groups. HACMP defines a standby cluster as one with a single host (called the active host) which owns all cluster resources. This procedure cannot be used in clusters utilizing rotating or concurrent resource group clusters.

1 Repeat the following steps for all hosts participating in a cluster:

   a Verify that cluster services are operating on all hosts.

   b Download and execute the following script: ftp://ftp.emc.com/pub/symm3000/aix/HACMP/ha4.3/convert.

   Execute the script to perform **emc_cfgmgr.sh** followed by **powermt config**.

   c Configure the cluster to execute the **convert** script as a pre-event to the node_down_remote event.

   d Use appropriate cluster commands to synchronize the cluster topology across all hosts in the cluster.

2 Repeat the following steps for all passive (not owning cluster resources) hosts participating in a cluster:

   a Install PowerPath software. Do not execute the **powermt config** command.

   b Connect the new paths between the Symmetrix and the host.

   c Configure the cluster to execute the **convert** script as a pre-event to the node_down_remote event.

   d Stop all applications from using all volume groups except **rootvg**. Unmount all file systems that are residing on these volume groups. Execute the command **varyoffvg <volume_group>** for each volume group to deactivate it.

3 Perform the following actions on the active host of the HACMP cluster. Stop cluster services in takeover mode, causing the transfer of cluster resources to a passive host. During the failover process the **convert** script will be executed, causing the configuration of paths and hdiskpower devices. This action has to be performed for all hosts participating in the cluster.

4 The following actions have to be completed on the original active host:

   a Install PowerPath software. Do not execute the **powermt config** command.

b Connect the new paths between the Symmetrix and the host.

c Configure the cluster to execute the **convert** script as a pre-event to the node_down_remote event.

d Stop all applications from using all volume groups except **rootvg**. Unmount all file systems that are residing on these volume groups. Execute the command **varyoffvg <volume_group>** for each volume group to deactivate it.

e Fail-over cluster resources to this host.

5 Verify that cluster services are operating on all hosts.

6 The following actions have to be completed on the original active host:

a Configure the cluster to remove the **convert** script as a pre-event to the node_down_remote event.

b Use appropriate cluster commands to synchronize the cluster topology across all hosts in the cluster.

CONCLUSION

EMC PowerPath software is an important component in total enterprise storage solutions. It has many similarities to AutoPath software, which supports HP XP as well as Hitachi storage devices. Both packages have similar features, which are compared in Figure 1. As we can see, the advantage of PowerPatch is its ability to provide more precise fine-tuning of load balancing to the user of the storage device. On the other hand, AutoPath has better integration with HACMP and better statistics-reporting abilities.

REFERENCES

1 *PowerPath for UNIX Concepts and Facilities Guide*, P/N 300-999-265, EMC Corporation

2 *PowerPath for UNIX Installation Guide*, P/N 300-999-266, EMC Corporation

	Auto Path	PowerPath
Non-disruptive installation	Latest versions only	Latest versions only
Automatic failover	Yes	Yes
Automatic on-line recovery	Yes	Yes
SMIT integration	Yes	Yes
Support for round-robin balancing policy	Yes	Yes
Support for number of I/O request-based load balancing policy	Yes	Yes
Support for size of I/O request-based load balancing policy	No	Yes
Support for path priority-based load balancing policy	No	Yes
Failover only support	Yes	Yes
Ability to disable both balancing and failover	No	Yes
Support for concurrent HACMP	Yes	Yes
Support for non-concurrent HACMP	Yes	Yes
Support for non-disruptive installation under non-concurrent HACMP	Yes	Yes
Support for report of adapter/device I/O statistics	Extensive	Limited

Figure 1: Comparing features of AutoPath and PowerPath

3 *PowerPath for UNIX Administration Guide*, P/N 300-999-267, EMC Corporation

Alex Polyak
System Engineer
APS (Israel)

AIX news

IBM has added updated software to its AIX V5.1 Expansion Pack, including Network Authentication Service V1.2.0.1, Netscape Communicator 4.79, and Encryption for SecureWay Directory V3.2.2.

The company also announced new software for its Bonus Pack, including AIX Developer Kit, Java 2 Technology Edition, V1.3.1, 32-bit version for POWER. This includes the tools to build secure Java applications for encryption. Also included is Open Secure Shell 2.9.9, Open Secure Sockets Layer (OpenSSL) V0.9.6b, which can be installed from the AIX Toolbox for Linux applications CD.

Updated software includes AIX Developer Kit and J2TE 1.3.1 64-bit version for POWER.

For further information contact your local IBM representative.
URL: http://www.ibm.com/servers/aix.

* * *

IBM has announced Version 2.1 of its WebSphere Business Integrator for AIX, for creating, executing, and managing business processes.

The software speeds implementation of new business processes, which, once implemented, help reduce the time, complexity, and cost of change. The resultant framework includes a model for EAI B2B integration, processes that span applications and enterprises and are managed as a single element, life-cycle management across an entire process, IBM's adapter framework, definition of process and data-level interactions between trading partners, and the ability to use WebSphere Studio Business Integrator Extensions to build reusable process templates.

For further information contact your local IBM representative.
URL: http://www.ibm.com/software/websphere.

* * *

4Front Technologies has announced OSS/AIX Version 3.9.6c, which now supports AIX 5.1.

There is also support for Sound Blaster PCI128, CMedia CMI8738, and ForteMedia FM801, plus support for all ISA PnP soundcards. The new version also provides improved installation for the drivers and automatic soundcard detection.

For further information contact:
4Front Technologie, 4035 Lafayette Place, Unit F, Culver City, CA 90232, USA.
Tel: (310) 202 8530.
URL: www.opensound.com.

* * *

IBM has announced its Migration Utility for z/OS and OS/390, allowing sites to create standard COBOL reports using Computer Associates' Easytrieve Plus language without Easytrieve Plus installed, and which run in place of the Easytrieve run-time interpreter. Easytrieve applications can be ported to any platform with a supported COBOL compiler, including AIX.

For further information contact your local IBM representative.
URL: http://www.ibm.com/software/ad/migration.