78

AIX

April 2002

In this issue

3 Input and output redirection
15 AIX application management
25 Examples of shell scripts
35 Find the smit fast path to what you want
40 SAN basics
43 AIX vulnerability
44 AIX news

update

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 1998 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 and Richard Watson

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.

Input and output redirection

You should be aware that in this article we discuss only how the input and output for commands can be redirected, not the various ways in which scripts can receive their input, which will be covered in detail in a future article.

REDIRECTING STANDARD OUTPUT OF A SCRIPT

As you are no doubt aware, most commands get their input from standard input, and send their output to standard output. By default, standard input is received from your keyboard, and, if a command requires data from standard input, it waits for you to enter information via the keyboard. When a command sends data to standard output, the data is normally sent to your terminal screen.

You can redirect the standard input and output of a shell script just as you can for any other command. The output that appears on your screen will be determined by how and where you redirect the output of commands contained within the script itself.

In order to understand how this works, you should create three 2 or 3-line text files; for the purpose of this example we will call them file1, file2, and file3. It may be easier to see what happens in the following examples, if one file contains only numbers, the second only lowercase letters, and the third only uppercase.

Create a shell script called mysort, and enter the following lines:

```
sort file1
sort file2
sort file3
```

Now run the shell script after making the file executable, and you will see a sorted version of file1, followed by a sorted version of file2, then the sorted file3. However, if you invoke mysort with the command:

```
mysort > all_three
```

the output will be sent to the file all_three, instead of to your screen.

Suppose you change the contents of mysort to:

```
sort file1
sort file2 > file2_sorted
sort file3
```

When you run this new version you will now see only the sorted version of file1, followed by the sorted file3. The output of the second command has been sent to the file file2_sorted.

If you now redirect the output of mysort with:

```
mysort > just_two
```

you will see that the outputs of the first and third commands go to just_two, and that of the second command is sent to file2_sorted. In other words, if you redirect the input or output of a command within a shell script, then regardless of how the script is invoked, the input or output of that command will always be redirected as specified within the script.

REDIRECTING STANDARD ERROR

Commands normally send their error messages to a third data stream, standard error, which is a data stream much like standard output, and can also be directed to a file instead of the screen. Programs usually write their regular output to standard output, and their error messages to standard error.

Because standard output and standard error are entirely separate data streams, data that a command sends to standard error appears on a user's screen even when standard output has been redirected.

Normally you don't redirect a command's standard error because you want to see error messages displayed on your screen as and when they occur. This is particularly important if you attempt to run a program and it is unsuccessful. If you had redirected the standard error then you would not know that the command had not executed until you examined the output file.

File descriptors

There are occasions, however, when it is desirable to redirect standard

error. For example, the C compiler, cc, may write many lines of warnings and error messages to standard error if the program you attempt to compile has many syntax errors. Examining these messages carefully while looking at your program can best be done if you have saved these messages in a file rather than displaying them on the screen.

For example, to compile a program called prog.c and send any error messages to a file named errors, enter:

```
cc prog.c 2> errors
```

In general, the notation 2> *filename* is used to redirect a command's standard error; spaces are not allowed between the 2 and the > character, although they are allowed between the > and *filename*. The 2 preceding the > is a file descriptor which stands for standard error.

There are also other file descriptors associated with standard input and standard output. These are 0 for standard input, and 1 for standard output.

You can redirect standard output with the notation:

```
1> filename
```

or:

```
> filename
```

If you want to redirect standard output and standard error to the same file, there is a special way of doing this, which is achieved by using:

```
2>&1
```

You can think of the notation 2>&1 as meaning, send standard error to the same place that standard output is currently going to. Similarly, 1>&2 redirects standard output to the same place as standard error. Any command such as:

```
command > filename 2>&1
```

will have both its standard input and standard output sent to *filename*.

If in the above example you reversed the order, namely:

```
command 2>&1 > filename
```

then standard error would be directed to the same place that standard output is currently connected, which would be the terminal screen since the redirection of standard output to *filename* occurs on the command line only after the initial redirection of standard error.

When some commands are executed, messages are sent to standard error which are merely advisory messages rather than an indication of a particular error condition. For example, when you run the find command as an ordinary user, many annoying messages will be displayed on your terminal stating that you do not have access to a particular directory.

In such a situation it is not necessary to read the messages since they merely prevent you from actually viewing the pathname of the file you are looking for, which can become lost amongst the large number of warning messages displayed on your screen.

Messages such as these can be discarded, and this can be achieved by redirecting standard error to the /dev/null file, which discards all data that is sent to it:

```
2> /dev/null
```

Turning off standard error

An alternative to redirecting standard error to /dev/null is to turn it off completely. To do this the command should be of the form:

```
command 2>&-
```

This particular format can be used to turn off standard output with:

```
command >&-
```

and standard input with:

```
command <&-
```

You can turn on standard input, standard output, or standard error, once they have been turned off, by issuing the commands:

```
exec < /dev/tty    turns on standard input
exec > /dev/tty    turns on standard output
exec 2> /dev/tty   turns on standard error
```

The **exec** command is covered in greater detail later in this article. Although the above discussions have been directed at entering commands from the command line, they are equally applicable for use in shell scripts. Turning off standard error is frequently used to get rid of unwanted error or warning messages, for example when you want to set the output of a command to a variable:

```
FILELIST=$(find . -name \*log 2>/dev/null)
```

This will create a variable containing a list of files ending in 'log', but will not contain any of the usual find warning messages which we would also expect to see in the list if we had not redirected standard error. The construction **$(. . . .)** is a form of command substitution and will be covered in greater detail in a future article.

USING PIPELINES IN SCRIPTS

When the shell sees a command of the form:

```
command_1 |
```

it knows that the command is not complete; if you are entering commands from the command line, the shell will respond with the secondary prompt until you type in the rest of the pipeline. If you enter such a command in a script, you can continue with the pipeline on the next line.

For example, if you enter **who |** on the command line, then the rest of the command, **grep *userid***, can be typed at the secondary prompt. In a shell script, however, this may appear as:

```
who |
grep userid
```

In general, any pipeline consisting of any number of commands may be split across several lines, provided each new-line character occurs after the '|' symbol and before the next command in the pipeline. To break up a pipeline anywhere else, a '\' must precede the new-line character, since the shell normally treats the new-line character as a command terminator. A new-line character following the '|' is a special case, because there must always be a command following '|'.

REDIRECTION USING EXEC

The **exec** command can be used to rename or create new file descriptors. To change standard input from the keyboard to a file, you can use the command:

```
exec < file
```

The commands to be executed will now be read from *file*, which should contain a list of commands. After **exec** has finished executing the commands, it will return to the parent of the process, which may result in logging you off. Similarly, to change output redirection from the terminal to a file, use the command:

```
exec > file
```

Once the redirection is no longer required, reassign the input/output back to the keyboard/terminal using the **exec** command with the virtual file /dev/tty:

```
exec < /dev/tty
```

or:

```
exec > /dev/tty
```

To see how this works, enter the commands:

```
exec > file4
cat file1
cat file2
ls -l
exec > /dev/tty
```

You will note that each time you enter a command, no output is displayed on the screen since it is all sent to file4. View the contents of file4 and you will see that the output from each of the commands has been appended to file4.

So when do we use **exec** in shell scripts? Quite often it is used within the .profile file of users who are required to run a particular application, but do not require a command prompt, either for security reasons or because the user wouldn't know what to do if confronted with the prompt. Usually the last entry in .profile is something like:

```
exec application
```

so that when the user exits the application they are immediately logged off. This assumes that there is no possibility of the user 'shelling out' from the application, otherwise there is not much point in placing the line in the user's profile.

A further use of **exec** in scripts is when, for example, you want to redirect the whole of standard output or standard error for the script to log files. You can achieve this by placing lines similar to the following at the top of the script:

```
exec > log
exec 2> errlog
```

These commands are suitable provided that you want all script output to be sent to the specified files. If this is not the case then you may have to redirect output for specified commands to other files.

Command line control using xargs

While on the topic of I/O redirection, let us consider a very useful command, **xargs**, which is used to manipulate the output from commands and is used extensively both from the command line and also within shell scripts.

When the system runs a command, the kernel provides a buffer to preserve the command and all its arguments, and under certain circumstances it is not difficult to exceed the buffer limit. For example, when you are using filename generation characters to specify files to be run against a command, the buffer size will be easily exceeded if a large number of filenames is generated, each with a long pathname.

The **xargs** command can be used to get round this upper limit, and it can be used to generate a series of commands from a list of filenames, or from other information supplied to it. **xargs** will read successive lines from its standard input, break the lines into words separated by spaces, reassemble the words into commands, and then pass the generated commands one by one to the shell for execution. It will always discard empty lines.

The command lines generated by **xargs** differ from those generated by the shell with wildcards in one important respect: **xargs** will not build

a command line longer than some reasonable upper limit. If the command line is likely to exceed the length limitation, **xargs** will distribute the filename list over two or more generated command lines.

As an example, **xargs** can be used to provide a handy check for common filenames located in two directories. For example:

```
cd dir1; ls | (cd ../dir2; xargs ls 2>&-)
```

The list of filenames generated by the first **ls** command is converted by **xargs** into a series of explicit **ls** commands, which are executed in the other directory. Files that cannot be found result in the printing of an error message, and turning off standard error will discard these messages. The only output seen is the list of files that are common to both directories.

There are a variety of options available to **xargs** for controlling command line formatting. Suppose we wanted to copy an automatically generated list of files to a specified directory. The normal command to do this is:

```
cp file1 file2 file3 . . . . dirname
```

Unfortunately this does not match the **xargs** style of generating commands. The solution, however, is to use the **-i** option:

```
ls dir1 | xargs -i% cp % dir2
```

This causes **xargs** to substitute one word from standard input for each occurrence of the character % in the command line; no spaces are allowed between the **-i** and the %. This construction results in the execution of a series of commands:

```
cp file1 dirname
cp file2 dirname
.
.
```

You can use any string or character following the **-i** option. For example, you could use:

```
ls dir1 | xargs -iFILE cp FILE dir2
```

The default string is a pair of braces, and the following command takes advantage of this fact:

```
ls dir1 | xargs -i mv {}.log {}.old
```

Another useful option to **xargs** is the **-p** option, which can be effectively used only from the command line. This option causes **xargs** to prompt for permission to execute each generated command before passing the command to the shell. This could be used in the copying example above, where the list generated may contain directory names so that you can be given the option not to run the command against a directory name.

The **-n** option also allows you to limit the number of arguments placed on the generated command line. For example:

```
xargs -n2 diff <<END
file1 file2 file3
file4 file5 file6

END
```

This construction (explained in the next section) runs a series of **diff** commands, each containing two filenames (generated by the **-n2** argument):

```
diff file1 file2
diff file3 file4
diff file5 file6
```

HERE DOCUMENT

In the above example, the <<END and END words define a 'here' document, which uses the text entered between the two END keywords as standard input for the **xargs** command, or any other command for that matter. The keywords do not have to be END, but can be any string or sequence of characters, inserted either directly into the here document, or by using command substitution; END and EOF are commonly used keywords.

When you run a command such as the previous **xargs** command from the command line, you will be prompted with the secondary prompt until such time as you enter the terminating second keyword.

If you run such a command in a shell script, you should ensure that the first keyword is on the same line as the '<<'. The files, or other string

of characters, on which you want to run the command are then placed on any number of following lines, and the terminating keyword is placed on a line of its own, unless there is redirection.

If you intend to redirect the output of the command you are using with **xargs**, and at the same time use the here document format, then the redirection in a shell script can be done as follows:

```
xargs -n2 diff > outfile <<END
file1 file2 file3 file4 file5 file6
END
```

or:

```
xargs -n2 diff <<END > outfile
file1 file2 file3 file4 file5 file6
END
```

A common use of the here document in shell scripting is to use **cat** to **echo** strings to the screen. For example:

```
cat <<%
string1
string2
.
.
%
```

It is not necessary to enter only text strings within here documents; the output of commands can also be supplied. For example:

```
cat <<% >> logfile
`date`
`who`
%
```

In this example the output generated from the **date** and **who** commands is supplied by the here document as input to **cat**. This type of here document uses backquotes (`) for command substitution. This is covered in detail in a future article.

You may also see strange-looking here document constructions in shell scripts. For example:

```
restore -tvf /dev/rmt0 <<EOF

EOF
```

You may find something like this in a script which creates a system back-up, and then tries to verify the back-up after rewinding the tape and skipping to the tape block containing the backed up files; if we encounter no error conditions during the **restore** (in this case just a read) then we assume the back-up was successful. The seemingly blank line actually contains a <CR>, which is required because the restore command asks you to mount volume 1 and press *Enter* so that the here document supplies the carriage return to the command and it then continues without user intervention.

Here documents are frequently used to provide input for commands or scripts that are run by cron. Since cron often runs commands at times when a user would not normally be present to enter the input, the here document allows the input to be supplied automatically.

As a further example, suppose an application is required to be started without user intervention, and in order to start the application a login is necessary. This login may need both a userid and password to be entered before it can be started. This can be done with a here document such as:

```
app_start << END
userid_string
user_password_string
END
```

The userid and associated password must be entered in the script as above and will automatically be supplied to app_start. When app_start first asks for the login name, it will be obtained from the first line of the here document. Similarly, the password will be obtained from the second line. In situations like this, the permissions on the script must be carefully controlled to prevent unauthorized access to the file.

This type of script can be used to allow logins without user intervention. Some applications, however, insist they receive standard input from the keyboard so that the userid and password cannot be supplied from a here document; **rlogin** is just such an example. In many cases the only way to find out whether you will be successful is to try it and see!

You should be aware that here documents take as input all the characters contained on each line between the keywords, and this may

give you unpredictable results. For example, suppose you had a shell script and you wished to indent the here document to make the script easier to read. If you had a construction such as:

```
if . . .
then

app_start <<END
userid_string
user_password_string
END

fi
```

then both the *userid_string* and *user_password_string* lines would include in the input to *app_start* the spaces or tabs from the beginning of the line, and this would mean a failed login. You can overcome this by using the construction:

```
if . . .
then

app_start<<- END
userid_string
user_password_string
END

fi
```

The <<- (the last character is a minus sign) tells the system to ignore tabs at the start of the line and thus allows formatting which is more readable.

You should be aware that if you use this construction then the line can start with either multiple tabs or a single space. Since you cannot often distinguish between spaces and tabs during a vi editing session (unless you have used :set list), you must exercise some degree of care to ensure that you get the correct input to your command.

Tonto Kowalski
Guru (UAE)

AIX application management

FEARS AND NEEDS

Once I was asked to come and help upgrade an AIX machine. “It is a standard job”, I thought. But coming to the data centre the IT manager told me to come back another day. The administrator who managed the application running on that particular machine had had an accident. No other person or documentation could help me stop and start it. His back-up administrator had left the company some weeks before.

Imagine that for some reason the application *had* stopped. In that data centre they would have called the hotline to the software company that sold them the application. This means external help is needed. This is no way to deal with a business critical application (see also Alan Prangley, Safetynet, *How to keep disaster at bay*, published in *Help Desks, Call Centres, and The Future*, Xephon, July 1999).

A day lost by restarting a business-critical application may be a nail in the coffin of your company. Therefore the administrator should provide some documentation for managing his basic tasks. There are many caveats in case of error or during certain complex operations, but this is no reason for non-provision of simple basic documentation. Consider the alternative – to attempt a restart without knowing anything.

HACMP helps you to restart your applications on another AIX machine if the primary one fails. Therefore it also solves the problem of stopping and restarting the application. Since nearly all software can be incorporated into a HACMP cluster, it proves that using such scripts is not impossible (Ole Conradsen et al, *HACMP/ES customization examples*, IBM Redbook 2000 cf Appendix A).

CREATING DOCUMENTATION

The main objective of a data centre is to provide continuous service. In order to achieve this, the data centre will need to provide:

- A (searchable, electronic) diary of incidents and their solutions.

- Mechanisms and precautions within the system to avoid incidents.
- Mechanisms within the system to allow quick recovery.
- Printed documentation dealing with events outside the system concerned.
- Organizational structures to support decisions and actions in critical situations.

The system administrator's job is to provide an appropriate recovery procedure when some application or service fails. There are several, often quite different, steps and solutions needed to reach this single objective. It is a challenge to structure and bundle the known actions for keeping the application or service available. Moreover all findings should be documented and accessible where they are needed.

Consistency problems between documentation and application handling can be avoided when both tasks are unified, ie the application handling is done, for example, by scripts that provide good documentation as well. On the one hand the documentation will be complete since the script wouldn't run otherwise, and on the other hand the scripts may be better understood since they are intended to document the system.

There are no clear conventions on where to put these documented scripts; some prefer to put them into */usr/local/bin*, others prefer to put them with the application or the middleware installation directory */usr/sbin/cluster/local*, or elsewhere. I think this is a question of philosophy and can be solved on another level: since all AIX administrators use SMIT it is probably a good idea to use SMIT for starting your applications as well, independently of HACMP.

MANAGEMENT SCRIPTS

Lessons from HACMP

Besides the handling of topology and resources, HACMP needs start and stop scripts for each application (*HACMP 4.4.1 Installation Guide*; *Chapter 12: Configuring Cluster Services*). Those scripts (*HACMP/ES Customization Examples* cf Section 7.1.2) need to support an unattended startup and shutdown of each application.

HACMP launches an application server as part of the HACMP startup or shutdown sequence. The application server executes a predefined script to start or to stop an application. At this point, the application is started or stopped in the background without any intervention. The non-interactive startup or shutdown is important to ensure that the application is launched or stopped with the fail-over sequence to minimize the startup or fail-over time, thus minimizing down time.

These scripts will improve with frequency of usage. The accumulation of experience of the system administrator is best hard-coded into these scripts. If there is no simple way to work around known difficulties automatically, it is always a good idea to state them as a comment and describe a likely way to handle them. It is always a good idea to use these scripts during everyday work.

Using SMIT

The application registration script in this article allows you to add a menu to your SMIT's top level menu *Applications*, which is initially empty. Within this article the application registration script will not be discussed, but the input parameters it requests will be. They are:

- Application Name – the name of the instance that should be stopped or started. It should, for example, not be called SAP, but SAP instance P01. Otherwise you might have difficulties distinguishing the instances.
- User Name – the name of the account that should be used to start and stop the application. If a login to this account fails or the user is unknown it will be set to root.
- Start Script – a shell script that is already discussed in the previous section as well as in the next one. Only existing executable scripts will be accepted.
- Stop Script – a shell script that is already discussed in the previous section as well as in the next one. Only existing executable scripts will be accepted.

When adding applications to the menu, they should be placed in an appropriate order. For an inexperienced administrator, it is then easy to

start task by task and to stop them in reverse order. Since the script from this article sorts the entries by the time of registration, you should consider this when adding the applications.

During the registration, a sequence of commands is written to a file, which can be used to remove the items from your SMIT menus. You should not lose it, since removing the items without knowing the IDs may result in some work in retrieving the information by executing *odmget* on *sm_cmd_hdr* and *sm_menu_opt*. You may look for the strings appearing in your menus. Be sure you make a back-up and ensure that you know what you are doing before going ahead (see Siegert, Andreas: *The AIX Survival Guide*, Addison-Wesley, 1996).

A shell program to enable the starting and stopping of applications using SMIT:

```
#!/usr/bin/sh
###############################################################################
##
## Description:
##    This shell script adds a submenu with start and stop option to
##    your SMIT's top level menu APPLICATIONS which is initially empty.
##    (This script is not originated or officially supported by IBM.)
##
##      You need to provide some parameters:
##      APPS:   the exact name of what application you want to control
##      USER:   the user to execute the start and stop scripts
##      STOP:   the script to stop the application
##      START:  the script to start the application
##
###############################################################################

###############################################################################
#
# Some example settings for those people reading this file
#
###############################################################################
USER=p01adm
APPS="SAP instance P01"
STOP=/home/p01adm/stop_sap
START=/home/p01adm/start_sap
###############################################################################
#
# Getting some parameters from the administrator
#               (admin=the one calling this script)
#
```

```
###############################################################################
cat <<***

        You are currently executing a script to add
        a start and a stop script to your SMIT menu
        called "Applications" (top level) for one of
        your applications.  Please enter the data now.


***
/usr/bin/echo "\tthe name of the application > \c"
read APPS
until [ -x $START ]
do
    /usr/bin/echo "\tthe fqpn of the start script > \c"
    read START
done
until [ -x $STOP ]
do
    /usr/bin/echo "\tthe fqpn of the stop  script > \c"
    read STOP
done
/usr/bin/echo "\tthe user logname to execute them > \c"
read USER
#/usr/bin/echo "\tthe fastpath you want to suggest > \c"
#read NAME
###############################################################################
#
# Some strings must be prepared before writing stanza file
#
###############################################################################
su - "$USER" -c /usr/bin/date >/dev/null 2>&1
                                                # see whether a "su" works
if [ $? -ne Ø -o -z "$USER" -o "$USER" = "root" ]
then
        # USER wasn't named or is root anyway; just execute the script
        EXEC=""
        USER="root"
else
        # ensure the user is set correctly before start script
        EXEC="/usr/bin/su - $USER -c "
fi
if [ -z "$NAME" ]
then
        # The value of NAME is your fastpath !!!
        # set NAME to the last token if it is most specific
        #NAME=$(echo "$APPS" | sed 's:[^ ][^ ]* ::g')
        # set NAME to the acronym, which is likely to be duplicate
        #NAME=$(echo "$APPS " | sed 's:\(.\)[^ ][^ ]* :\1:g')
```

```
        # this one is more reliable, but more likely a "slowpath"
        NAME=$(echo "$APPS" | sed 's: ::g')
fi
ODMDIR=/usr/lib/objrepos
SEQN=$(date +%Y%m%d%H%M%S)

cat <<***

        Your parameters are
        Name of the Application:        $APPS
        Application start script:       $START
        Application stop  script:       $STOP
        User to call both scripts:      $USER

        Please press ENTER to extend smitty or press CTRL-C to abort
...
***
read X
echo "\t\t\t... adding your menu now ... \n"
#############################################################################
#
# Writing stanza file and add it to ODM in file:/usr/lib/objrepos/
#
#############################################################################
cat <<*** >/tmp/smitty.add
sm_menu_opt:
        id_seq_num = "$SEQN"
        id = "apps"
        next_id = "$NAME"
        text = "Manage $APPS"
        text_msg_file = ""
        text_msg_set = 0
        text_msg_id = 0
        next_type = "m"
        alias = ""
        help_msg_id = ""
        help_msg_loc = ""
        help_msg_base = ""
        help_msg_book = ""

sm_menu_opt:
        id_seq_num = "$(/usr/bin/expr 1 + $SEQN)"
        id = "$NAME"
        next_id = "start$NAME"
        text = "Start $APPS"
        text_msg_file = ""
        text_msg_set = 0
        text_msg_id = 0
        next_type = "d"
```

```
        alias = ""
        help_msg_id = ""
        help_msg_loc = ""
        help_msg_base = ""
        help_msg_book = ""

sm_cmd_hdr:
        id = "start$NAME"
        option_id = "Opts$NAME"
        has_name_select = "n"
        name = "Start $APPS"
        name_msg_file = ""
        name_msg_set = Ø
        name_msg_id = Ø
        cmd_to_exec = "/usr/bin/su - $USER -c $START "
        ask = "n"
        exec_mode = ""
        ghost = "y"
        cmd_to_discover = ""
        cmd_to_discover_postfix = ""
        name_size = Ø
        value_size = Ø
        help_msg_id = "Ø"
        help_msg_loc = ""
        help_msg_base = ""
        help_msg_book = ""

sm_menu_opt:
        id_seq_num = "$(/usr/bin/expr 2 + $SEQN)"
        id = "$NAME"
        next_id = "stop$NAME"
        text = "Stop $APPS"
        text_msg_file = ""
        text_msg_set = Ø
        text_msg_id = Ø
        next_type = "d"
        alias = ""
        help_msg_id = ""
        help_msg_loc = ""
        help_msg_base = ""
        help_msg_book = ""

sm_cmd_hdr:
        id = "stop$NAME"
        option_id = "Opts$NAME"
        has_name_select = "n"
        name = "Stop $APPS"
        name_msg_file = ""
        name_msg_set = Ø
```

```
        name_msg_id = Ø
        cmd_to_exec = "$EXEC$STOP "
        ask = "n"
        exec_mode = ""
        ghost = "y"
        cmd_to_discover = ""
        cmd_to_discover_postfix = ""
        name_size = Ø
        value_size = Ø
        help_msg_id = "Ø"
        help_msg_loc = ""
        help_msg_base = ""
        help_msg_book = ""

***
/usr/bin/odmadd /tmp/smitty.add
return=$?
/usr/bin/rm -f /tmp/smitty.add
##############################################################################
#
# Allow easy removal of the new menu entries
#
##############################################################################
cat <<*** >$HOME/not$NAME
ODMDIR=/usr/lib/objrepos
odmdelete -o sm_menu_opt -q id_seq_num="$(/usr/bin/expr Ø + $SEQN)"
odmdelete -o sm_menu_opt -q id_seq_num="$(/usr/bin/expr 1 + $SEQN)"
odmdelete -o sm_menu_opt -q id_seq_num="$(/usr/bin/expr 2 + $SEQN)"
odmdelete -o sm_cmd_hdr -q id="start$NAME"
odmdelete -o sm_cmd_hdr -q id="stop$NAME"
***
chmod a+x $HOME/not$NAME
echo "\n\tThe menus will be removed from your menus by \"$HOME/
not$NAME\"."
##############################################################################
#
# Our work is done now ; terminate happily
#
##############################################################################
echo "\tThe fastpath to the applications menu is \"apps\"."
echo "\tThe fastpath to your application is \"$NAME\"."
echo "\tThe fastpath to stop your application is \"stop$NAME\"."
echo "\tThe fastpath to start your application is \"start$NAME\"."
exit $return
```

Start and stop scripts

The constraints for the start and stop script are similar to the ones given by HACMP, eg:

- There must be no prompting for user input.
- It should be tolerant of inappropriate execution.
- It should handle a second attempt to start an application after failure for some condition which was subsequently corrected.

Please note that there is no restriction on displaying information during the execution of the start or stop scripts. If one fails, the analysis is simplified by reading a rather verbose sequence of messages rather than only a concluding note stating success or failure.

AUTOMATION

The previous discussion focused on two aspects of manual application handling. These aspects may be extended to total automation:

- Automated startup
- Automated stopping.

Automated startup

For the startup process of a computer, there are different philosophies. If you follow the 'Windows'-philosophy, you designate a single computer to a single application. Therefore you expect that restarting the *server* machine also means restarting the application. (*Server* originally described the communication part of application software.)

The 'multi-tasking' philosophy detaches the layer of starting the hardware from starting the software. Whether it is better to start applications automatically at boot time or manually later cannot be answered briefly and can depend on the exact situation. Nevertheless, the manual startup should *always* be implemented.

If an automated startup process is selected, there is usually one way to do it – adding the service to the standard startup sequence. Within the */etc/inittab* there are usually services started that are expected to be present all the time during system uptime. These should be insensitive to shutdown commands and not depend on resources, like network file systems or directories that are not on volume groups, that start later with autovaryon.

If the application is to be detached so it runs in the background it should be added to the */etc/inittab* by executing:

```
mkitab "myapp:2:once:/my/app/startup </dev/null >/dev/console 2>&1"
```

where */my/app/startup* should be the standard startup file for the application tagged *myapp*.

If the process to be started is robust, it may be started in foreground, and if it needs to be present all the time it may be started as respawning:

```
mkitab "myapp:2:respawn:/my/app/startup </dev/null >/dev/null 2>&1"
```

The *respawning* action ensures that the program will be restarted each time it leaves the process table. It *respawns* too rapidly if the script terminates while the task has not yet been performed. If you are unfortunate, the machine's memory runs low, the CPU gets overloaded, and you have to correct the script by booting in maintenance mode. Be careful to avoid such unpleasant situations.

Putting the application startup into */etc/rc.tcpip*, or similar, is another way to start it, but, in principle, this is another step in cascading the start script. It increases the complexity.

Automated stopping

All applications *must* be stopped before a system shutdown. Otherwise information will not be treated as the application thinks necessary. The standard AIX method is adding the stop script to */etc/rc.shutdown*. This file does not exist by default – create it and add all stop scripts in the correct order, which is usually the reverse of the one for startup.

In all situations where a proper shutdown is possible the script */etc/rc.shutdown* will be executed. There are situations you may not be aware of, for example if your UPS batteries signal that they are running out of power, or the system overheats because of stopped fans. It is rather unlikely that you will be able to catch all those events manually. The *rc.shutdown* will handle that for you.

CONCLUSION

This document shows the need for a standard way to manage at least your business-critical applications. It is intended to help you to increase your computer centre's availability, where availability means the usability of applications. It also recommends a standard way of dealing with your applications by focusing on the most important parts. Therefore only starting and stopping is described here. Some technical support is provided to achieve this objective.

Andreas Neuper
PROFI Engineering Systems AG (Germany)

Examples of shell scripts

The best way to start to learn when and how to write shell scripts is to look at the following examples and try writing the scripts involved. The first example is a very trivial script and can be ignored if it is beneath your dignity to create it!

EXAMPLE 1 – SIMPLE ECHO SCRIPT

Often you can save yourself a lot of typing by creating a simple shell script. For example, for most ASCII terminal types the **clear** command will clear all characters from the screen. Some terminal types, however, may not support the **clear** command, and you can create a simple shell script named **cl** that will perform the same task.

One way to clear the screen is to use the **echo** command to send 24 new-line characters to your screen. The **echo** command interprets the pair of characters **\n** as a new-line character.

We use single quote marks around the argument to echo because the backslash, '\', has a special meaning to the shell. Since we want it to be passed to **echo** without being interpreted in its normal way by the shell, the quotes will protect everything within them from such an interpretation.

When you edit the cl file, enter the following:

```
#!/bin/ksh
echo '\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n'
```

Use the **chmod** command to make it executable and place it in your $HOME/bin directory. Thereafter, whenever you enter **cl**, the screen will be cleared.

This is obviously not the simplest or most desirable command to use in this particular shell script to achieve a clearing of the screen for terminal types that support the clear command, but the example merely serves to show you what can be achieved with a simple script.

You could, for example, have entered **clear** in the script (provided it is supported by your terminal) instead of the **echo** command, and this would have produced the same result. In any event, it would normally not be worthwhile creating such a file in order to save typing in three extra characters!

EXAMPLE 2 – COUNTING WORDS IN FILES

By their nature, Unix and AIX contain many complex commands with multiple command line options, which are often difficult to interpret, and almost impossible to remember in full.

The following script uses the **tr** command, which has particularly complex constructions for its command line arguments. It may take an age to learn these in order to run the command just once from the command line, and of course the next time you want to use it for the same purpose you have forgotten the construction. Scripts like the following make life much easier, at least temporarily!

Suppose, for example, you wanted to print both a list of each of the words contained in a file and the number of times each word was used. In order to create a script to do this, first create a text file named mytext containing the following two sentences:

```
Hello, I'm a simple text file.
I don't contain much text but I do contain a little.
```

What you now want to do is produce a list which calculates the number of times a word is used in the file and display the output in a format similar to:

```
2 a      1 file    1 little
```

```
1 but  1 hello  .
.
.
```

The strategy we will use is to break the file into a list of words, sort the list so that repeated words will be on consecutive lines, count the number of times each word is repeated, and finally produce a list of the words, each preceded by a count of the number of times it appeared.

The **tr** command is used to break up the file, one word per line, and the **sort** command is used to sort the list. The **uniq** command removes duplicate words and counts the number of times each word occurs. Finally the little used **pr** command is used to format the list.

The sequence of commands will look something like:

```
tr arguments < mytext | sort | uniq | pr
```

The format of the **tr** command is:

```
tr [options] string1 string2
```

and the command replaces each of the characters in *string1* with the corresponding character in *string2*. It does not change a word represented by *string1* with a word represented by *string2*; the translation is from a single character in the first string to the corresponding single character in the same position in the second string.

To first change all the uppercase letters in mytext to lowercase, you would enter the command:

```
tr  -A  '[A-Z]'  '[a-z]'  < mytext
```

The A-Z and a-z are shorthand notations for A to Z and a to z respectively, and must be enclosed in square brackets. When these abbreviations are used, they must also be enclosed in quote marks to ensure that the shell interprets the square brackets as boundaries for the shorthand notation, and not just as square bracket characters.

The **-A** option tells **tr** to perform all operations using the ASCII collation order. Depending on your current version of the operating system, if you do not use this option you may get some strange and unpredictable results when dealing with strings and characters.

To break up mytext, one word per line, we need to replace all spaces and punctuation marks with new-line characters. This can be achieved

with the following command:

```
tr -scA "[A-Z][a-z]'-" '[\012*]' < mytext
```

Try this command and note the output it produces.

The **-c** option (for complement) instructs **tr** to replace all characters not found in the first string, which in this case means all characters which are not uppercase letters, lowercase letters, apostrophes, or hyphens.

Note that the quote marks around the first string are double quotes. This is because the first string itself contains an apostrophe, which is equivalent to a single quote, and the only way to quote a single quote is by surrounding it with double quotes. More on quoting later.

The **-s** option (for squeeze) causes **tr** to avoid generating two characters from the second string in a row. In this case, the **-s** ensures that there will not be any blank lines in the output, which would occur if two or more new-line characters were generated.

Any character not contained in the first string, in other words any character that is not part of a word, will be replaced by a character from the second string. In this case we want to replace any character that is not part of a word with a new-line character, so the second string should contain only new-line characters.

A new-line character can be specified by using its ASCII Octal code, 012, preceded by a backslash. Since the number of characters in the first string must equal the number of characters in the second string, placing an asterisk after the new-line character and enclosing it in square brackets indicates to the shell that the new-line character should be repeated as many times as is necessary to pad out the second string to equal the length of the first.

Since the expression \012* is an abbreviation for a string consisting of many new-line characters, it must be quoted. Instead of using \012, you can use the alternative for new-line characters, **\n**.

You should now be able to see how the command works. It reads its standard input, mytext, and copies it to standard output, your screen, replacing all characters that are not part of a word with a new-line character.

Consider the following shell script which makes use of the **tr** and other commands:

```
#!/bin/ksh
tr -A '[A-Z]' '[a-z]' < mytext |
tr -scA "[a-z]'-" '[\Ø12*]' |
sort |
uniq -c |
pr -3 -t
```

The first **tr** command gets its standard input from mytext. The command translates all the uppercase letters in the file to lowercase and the result is piped to the second **tr** command. The conversion is necessary because **uniq** normally considers capitalized words to be different from uncapitalized ones and we want words such as 'The' and 'the' to be considered the same.

The second **tr** command substitutes a new-line character for each character that is not a letter, hyphen, or apostrophe. This command is slightly different from the one shown earlier because the input to this command will not contain any uppercase letters.

The resulting list of words is passed as input to the **sort** command, which processes them so that repeated words appear on consecutive lines.

The sorted list is processed by **uniq -c**, which outputs each unique line preceded by a count of the number of times that the line appeared. The list is sorted before being passed to **uniq** because **uniq** notices that a given line is repeated only if the lines are adjacent.

Finally, the output from **uniq** is piped to the input of **pr -3 -t**, which outputs a 3-column paginated version of the list without the usual header.

To understand how this shell script works, you need to see the output at each stage in the pipeline.

You can use an editor to produce your own copy of the script named word_count, but you will probably find it more helpful if you redirect the output of each command line to a series of files as shown below.

At the shell prompt, type the first command and instead of typing '|', redirect the output to temp1:

```
tr -A '[A-Z]' '[a-z]' < mytext > temp1
```

Now look at temp1 and see how it differs from mytext. You can do this with the command:

```
pr -mt mytext temp1 | pg
```

Next, enter the second line of text at the command line, taking the input from temp1, and redirect the output to a file called temp2:

```
tr -scA "[a-z]'-" '[\Ø12*]' < temp1 > temp2
```

Now compare temp1 with temp2 to see what the command did. Use a similar command to the **pr -mt** command above.

Then enter the following commands, comparing the input and output files of each command before running the next:

```
sort <temp2 >temp3
uniq -c <temp3 >temp4
pr -3 -t <temp4 >temp5
```

EXAMPLE 3 – USE OF PRINT AND PRINTF

We said in an earlier article (*Shell script basics*, *AIX Update* Issue 77, March 2002) that the intention was to use **print** rather than the **echo** command because of the former's more powerful formatting capabilities, which is not strictly true; it is the similar **printf** command, which is used for sophisticated formatting. Generally speaking, the functionality of **echo** varies from one Unix operating system to another, although this is not relevant here since we are only concerned with AIX. You should, however, get into the habit of using **print** rather than **echo**.

We used **echo** in the first example as a very rudimentary start to shell scripting, but the following example shows how **print** and **printf** can be used in shell scripts, and how useful they are for formatting text output.

Before creating the shell script in this example we first have to create a text file on which we are going to operate the script. To do this run the command:

```
lsvg -o | lsvg -i > vgs
```

This will create the vgs source file containing information on all the volume groups that are currently active (*varied-on*) on your system. View this file and you will see that it contains a series of lines for each

volume group, separated by blank lines. If your system has only rootvg active, then you will see only the single entry.

What we want to do is create a script which extracts from the source file the total size of the volume group and the amount of free space it contains, and then format this output with a heading. Initially we will be doing this only for rootvg, but as your skills develop we shall be able to do this for all volume groups that are active on your system.

Create the script called vgsizes containing the following lines:

```
#!/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)
print $VG: Total size = $TOTAL MB: Free space = $FREE MB
printf "%-20s %-15s %-15s\n" "Volume Group" "Total Size" "Free Space"
printf "%-20s %-15s %-15s\n" $VG "$TOTAL MB" "$FREE MB"
printf "%-20s %10d %15d\n" $VG $TOTAL $FREE
printf "%-20s %10.2f %15.2f\n" $VG $TOTAL $FREE
printf "%-20s %10d %15d\n" $VG "$TOTAL MB" "$FREE MB"
```

The first two variable assignments set the names of the volume group whose information is to be extracted from the source file, and the pathname of the source. We have hard-coded the volume group name whose details we want to extract in the VG=rootvg line, but you will see later how we can pass the name as an argument to the script and so give us greater flexibility in running the command.

In each of the lines setting the variables TOTAL and FREE, a series of commands is contained within the construction $(. . .). This type of construction is called *command substitution* and is discussed in detail in a future article. For the time being, all you need to know is that the variables are set to the final output of each group of commands.

If we look at these constructions we see that the first **grep** statement has the **-p** option. This searches in the source file for the volume group name contained within any *paragraph*, which is any group of lines from a blank line (or start of the file) to the next blank line (or end of the file). If the characters of the volume group name are not unique, then we may extract more than one paragraph. You will find that **grep**

-p is very useful for extracting whole stanzas from files, such as those contained in many AIX system files.

The next **grep** searches for the line in the output piped to it which contains the appropriate 'TOTAL' or 'FREE' characters. This should select the line containing, for example, the characters:

```
. . . TOTAL PPS:     479 (1916 megabytes)
```

The first **cut** command now extracts field two in its input line, where we have defined the fields to be separated by the delimiter '('. This will extract only the characters we are interested in:

```
1916 megabytes)
```

The **tr** command in the pipeline converts spaces to tabs since the next **cut** command into which its output is piped does not have a field delimiter specified with a **-d** option, and thus defaults to tabs. This last **cut** command extracts the total size (or free size as the case may be), which in this case is the characters '1916'.

The **print** command just gives standard formatting and is used here only to show you the output. The first **printf** command, however, produces formatted output, and the construction of the command is similar to that used in C programming. The formatting characters are contained within the first set of quotes after the command, and any subsequent character strings, which may or may not be contained within quotes, are the characters that are printed according to the formatting instructions.

In our example the **%-20s** characters tell us that its output is going to be a character string (the **s** part), 20 characters wide, and left-justified; you can right-justify by excluding the minus sign. If you require a group of characters containing spaces to be output to this field, then you must enclose these characters in quotes, as we have done with 'Volume Group'.

If in the formatting sequence you include characters that the system does not recognize as a formatting sequence, then the characters themselves will be printed. In our example we have included spaces to make it easier to read, but you do not have to do this. For example, **printf "%-20s%-15s%-15s\n"** is equally acceptable.

By default, **printf** does not print a carriage return at the end of each line

and so, if this is required, you must include one or more **\n** combinations in the formatting string. These may be anywhere in the line to produce carriage returns at selected locations, but in our example we have included them only at the end of the formatting sequence, since this is where we need the new line.

The remaining **printf** commands show you how the format can be modified. **%10d** expects digits in the field to be displayed and in this case will right-justify them. If in the string to be output you include any characters other than numbers, you will get error messages that are generated by the last **printf** command, because the character fields contain the unacceptable 'MB' characters separated by spaces from the actual digits.

The **%10.2f** format produces floating point output, ten characters wide, with two decimal places. There are a number of other formatting constructions used by **printf** and it is recommended that you view the manual pages to familiarize yourself with these.

When you have created and run this script, try experimenting with other formatting strings. Also add a further **printf** command so that underneath the 'Total Size' and 'Free Space' parts of the heading there are additional 'in MB' characters – perfectly lined up, of course!

One of the situations where you cannot use **print** is when you want to use something like **print -x**. This will most likely give you an error since the command will try to interpret the **-x** as a command line option. If you just want to print out these characters, then you must use either **echo** or **printf**.

EXAMPLE 1 – RECORDING MAIL SENT

The following example may not be particularly relevant in this day and age, where mail is rarely sent from one Unix user to another using standard Unix commands. However, the script is used to show what can be achieved, and as you acquire further shell programming techniques we will modify this script to make use of these.

Suppose you want to keep a copy of all electronic mail you send to your colleague Fred. More specifically, he is an untrustworthy person and you want to cover your own back by keeping a log file containing a copy of each memo you send to him! Each item in the log file should

be preceded by a line stating the date the memo was sent. If you were doing this from the command line you would use the following sequence of events each time you wanted to send a memo to the userid Fred.

You create a file, tmpfred, containing the memo, and record in your log file the time and date you are sending the memo by entering:

```
date >> fredlog
```

You then append a copy of the memo to the end of the log file:

```
cat tmpfred >> fredlog
```

You add a blank line to the log file to separate this entry from the next one:

```
print '\n' >> fredlog
```

You then mail a copy of the memo to Fred:

```
mail fred < tmpfred
```

Now that you have a copy of the memo saved in the log file, and you have sent a copy to Fred, you can remove tmpfred:

```
rm tmpfred
```

To make sure you understand what this procedure accomplishes, you may want to send two or three pieces of mail to yourself. Look at your log file periodically and make sure you use >>, not >, to redirect the output.

To maintain a complete log of the mail you send to Fred, you would have to repeat these six steps every time you wanted to send him a memo. This would probably become so tiresome that you would soon give up, so instead we will create a shell script that will perform each of these steps automatically.

Create the following script, name it fredmail, and enter the following lines:

```
#!/bin/ksh
LOG=fredlog
TMPFILE=tmpfred
vi $TMPFILE
date >> $LOG
cat $TMPFILE >> $LOG
```

```
print '\n' >> $LOG
mail fred < $TMPFILE
rm $TMPFILE
```

After making it executable, on entering the command fredmail you would start a vi editing session and create your memo. As soon as you exited from the editor, the shell script would continue with the next command, append the date, send the memo to your log file, mail the memo to Fred, and remove the temporary file.

Try creating this shell script for a user you know to be on the system, and send him or her a number of memos to test that it is working properly. Or, alternatively, send them to yourself.

You will notice that we have used variables for the name of the log file and the temporary mail file; we have used similar variables in the previous example also. This is good shell programming practice, and a habit you should quickly get into, since, if at some future date in this example you want to change either the log file name or the pathname of the temporary file then you only need to make a single change.

Tonto Kowalski
Guru (UAE)

Find the smit fast path to what you want

One of the really useful features of AIX is smit (the System Management Interface Tool), and its text-based version called smitty. This is a menu-driven facility for performing system-related tasks, and is always a good starting point for performing tasks for the first time. It will prompt you for whatever values are required for a given command and then build and run the command for you.

One of the things that makes smitty really great is that it is completely transparent and allows you to view the command it has built. So in the future you can run the command natively yourself without using smitty.

Another useful feature is a fast path option which will take you directly

to a specific command screen within smitty, saving you from having to navigate through its menus.

However, sometimes you know exactly what you want to do but you just can't find it within smitty. The following script gets round this problem by searching menus for a character string that you provide and returns the menus that match, along with their fast paths. Additionally, for the character string you enter, it will provide a list of menus you need to select in order to navigate to it.

```
#!/usr/bin/ksh
#
# Script: smitFastPath
# Author: Roger Wickings
# Aim: find a smitty fast path
#
awk="/usr/bin/awk"
basename="/usr/bin/basename"
grep="/usr/bin/grep"
odmget="/usr/bin/odmget"
sed="/usr/bin/sed"
sort="/usr/bin/sort"
tr="/usr/bin/tr"
ODMDIR="/usr/lib/objrepos"
ODM_CLASS="sm_menu_opt"
#
# Functions
#
showFastPath()
{
export ODMDIR
check='$odmget $ODM_CLASS |
$sed "s,\", ,g" |
$awk '$1 == "next_id" {print $3}' |
$grep -i "^$FASTPATH$" |
$awk 'END {print NR }' `
if test "$check" = "Ø"
then
showPossible
else
showTree
fi
return
}
showPossible()
{
$odmget $ODM_CLASS |
$sed "s,\", ,g" |
```

```
$awk '
$1 == "next_id" { nx = $3 }
$1 == "text" {
tx = $3
for ( i=4 ; i<=NF ; ++i ) { tx = tx " " $i }
print tx, "[" nx "]"
}
' |
$grep -i "$FASTPATH"
return
}
showTree()
{
$odmget $ODM_CLASS |
$sed "s,\", ,g" |
$awk '
$1 == "sm_menu_opt" { id = "" ; no = "" }
$1 == "id" { id = $3 }
$1 == "next_id" { nx = $3 }
$1 == "text" {
tx = $3
for ( i=4 ; i<=NF ; ++i ) { tx = tx "@" $i }
print id, nx, tx
}
' |
$awk 'NF==3 { print }' |
$sort |
$awk -v root=$FASTPATH '
BEGIN {

msub = Ø
osub = Ø
}
{
tid[msub] = $1
tnext[msub] = $2
ttext[msub] = $3
++msub
}
END {
for ( wsub=Ø ; wsub<msub ; ++wsub )
{
if ( tnext[wsub] != root )
{
continue
}
ssub = 1
stack[ssub] = wsub
start[ssub] = Ø
flag_up = "NO"
```

```
while ( ssub != Ø )
{
csub = stack[ssub]
cstar = start[ssub]
cid = tid[csub]
ctext = ttext[csub]
found = "NO" ### Look for next level ###
for ( tsub=cstar ; tsub<msub ; ++tsub ) ### tsub is temporary table
subscript
{
if ( tnext[tsub] == cid )
{
found = "YES"
flag_up = "YES"
start[ssub] = tsub + 1
++ssub
stack[ssub] = tsub
break
}
}
if ( found == "NO" )
{
if ( flag_up == "YES" )
{
flag_up = "NO"
++osub
print osub, ssub+1, cid, ctext
for ( psub=ssub ; psub>Ø ; –psub )
{
pout = stack[psub]
print osub, psub, tnext[pout], ttext[pout]
}
}
–ssub
}
} ### End of while loop ###
}
}
' |
$awk '
{
if ( $1 != current )
{
count = $2
print " "
}
current = $1
print substr(" ",1,3*(count-$2)) $4, "[" $3 "]"
}
' |
```

```
$tr "@" " "
echo
return
}
#
# Start of main processing
#
SCRIPT='$basename $Ø '
FASTPATH='echo "$1" | $sed "s,\",,g" '
showFastPath
exit
```

So, for example, if you wanted to find how to switch on async I/O you could run the following:

```
# smitFastPath async
Asynchronous I/O [aio]
Change / Show Characteristics of Asynchronous I/O [chgaio]
Remove Asynchronous I/O; Keep Definition [rmvaio]
Configure Defined Asynchronous I/O [cfgaio]
Trace Asynchronous I/O [trace_link]
Asynchronous Adapters [ttyadapters]
#
```

Now you can enter **smitty chgaio** to go directly to the smitty screen for switching on async i/o. Alternatively, if you wanted to navigate via the smitty menus, you could enter the following to find out which options to select:

```
# smitFastPath chgaio
__ROOT__ [__ROOT+__]
__ROOT__ [__ROOT__]
System Management [top_menu]
Devices [dev]
Asynchronous I/O [aio]
Change / Show Characteristics of Asynchronous I/O [chgaio]
#
```

This tells you that from the first smitty screen, which is System Management, you have to select the Devices sub-menu. Then from there you select the Asynchronous I/O sub menu.

Roger Wickings
Systems Programmer
FT Interactive Data (UK)

SAN basics

Like all system administrators for AIX systems, I have a varying array of roles to fulfil. In larger IT shops our roles are very specific to AIX, while in smaller organizations we tend to be 'Jacks of all trades'. Storage Area Networks (SAN) are one more area with which we will all need some level of familiarization. While there is no feature in AIX that addresses this directly, we, as sysadmins, will still be involved with SANs in one capacity or another.

In our environment, I recently ran out of available direct attachments from my (EMC) storage array to the Server environment. In order to get around this, and continue to exploit the available resources on the storage array, we had to introduce a switch in between the array and the servers. This sounded, and is, simple enough, but the temptation was there to let the term 'SAN' intimidate.

SAN IN A NUTSHELL

A SAN evolves, usually, from the need to provide portions of a storage resource to more than one server, or endpoint. The resource will usually be a storage array of the EMC Symmetrix type, IBM ESS ('Shark') type, or others. A simple illustration would be two point-to-point fibre connections – see Figure 1.

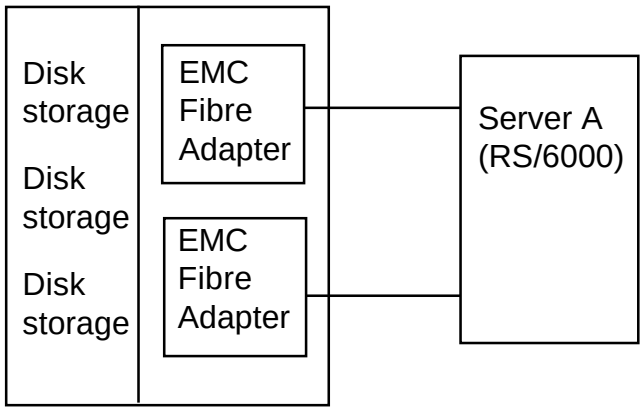

Figure 1: Two point-to-point fibre connections

STORAGE ARRAY

For illustration/example purposes only, I'm indicating that there is only one connection available from the fibre adapter to the Host Bus Adapter (HBA) on the RS/6000 server. This is not the case in real life. In this example we simply have two directly-attached connections from the server to the storage.

Now let's say we need to give access to some of the storage to Server A and some to Server B. Rather than install two more fibre adapters each time we need to add a server, a switch would be introduced. Typically this switch will be eight ports or more, four of which are in use, in this basic configuration – see Figure 2.

This is a SAN! Albeit, a very basic one. It is simply a closed network or 'fabric' of storage resources.

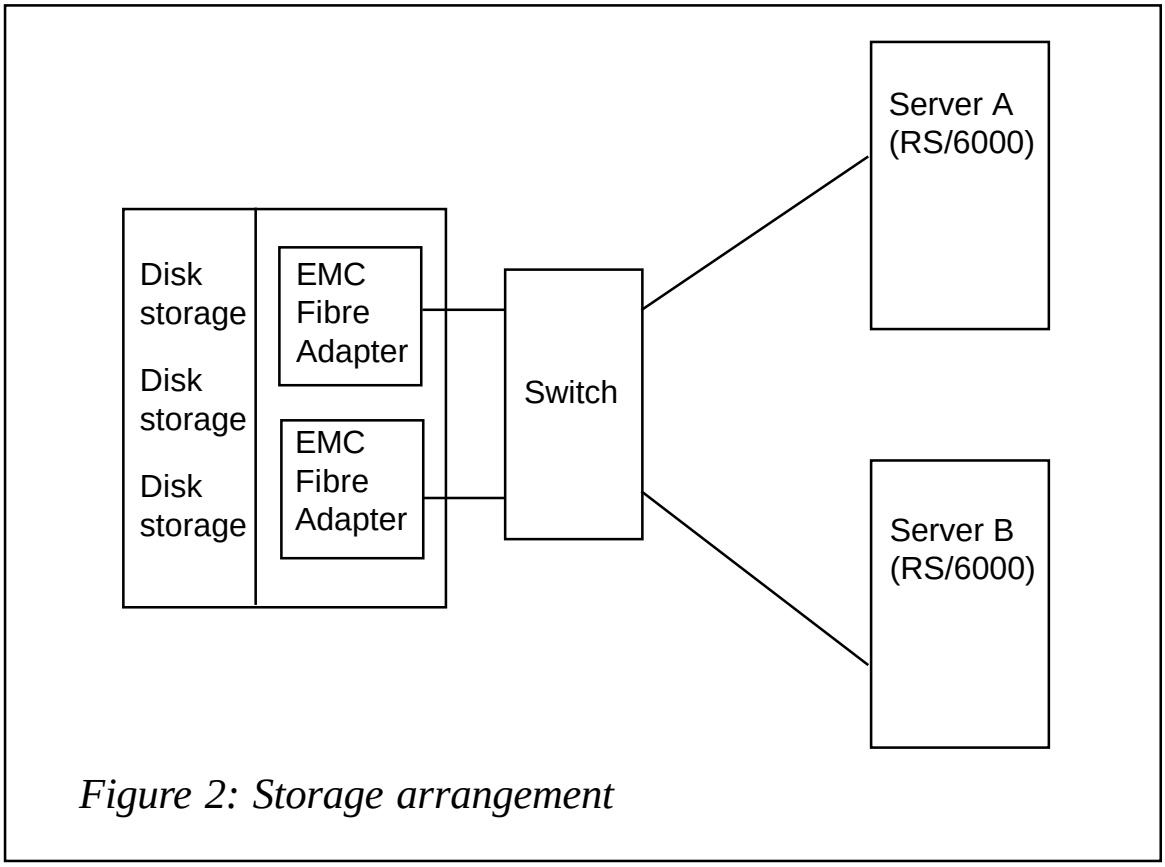

Figure 2: Storage arrangement

LOGISTICS – OVERVIEW

These steps are intended to be high-level, not specific, as that would require a much longer article.

In the first, directly-attached storage example, you would have:

- Allocated one or more storage devices to each fibre adapter.
- Run **cfgmgr** in AIX on the server.

And your disk(s) would be available for use.

In the second SAN example, there would be a few additional steps:

- Allocate one or more storage devices to each fibre adapter.
- Create a zone on the switch that tells it which storage device(s) each server (HBA) is permitted) to see.
- Run **cfgmgr** in AIX on the server.

There are a number of other details to attend to in the initial stages of installing the SAN that are outside of the scope of this SAN introduction. They mostly take place on the storage array and switch side to 'mask' LUNs (ie prevent them from being seen other than on the intended server), define 'World Wide Names' for storage resources (kind of like a MAC/NIC address for a disk resource), and things of that nature.

SAN TERMINOLOGY

Here are some definitions:

- Fabric – a network consisting of one or more devices.
- Domain – usually refers to one switch in the network/fabric.
- Node – a device in the network; server or storage.
- Initiator – a server node that initiates a request for data.
- Target – a storage device node that responds to an initiator.
- Port – a physical connection into a domain or node.
- WWN – World Wide Name, 64-bit unique address used to grant permissions between nodes.

FURTHER READING

As mentioned, I've intended this to be a high-level introduction to SANs, and hopefully to clarify a little of the mystique it has for some.

There is a wealth of material available for reading on the Internet.

Some links:

- http://www.gadzoox.com/san_library/whitepapers.html
- http://www.mcdata.com/servsupt/education/sanbasics/index.html
- http://www.brocade.com/san/white_papers.jhtml.

David Miller
Database Architect
Baystate Health Systems (USA)

AIX vulnerability

A security hole in software from IBM and Sun Microsystems could allow hackers to take control of servers running in many corporations. Researchers have uncovered evidence in Internet chat rooms that hackers have already started developing tools to take advantage of the vulnerability, which affects AIX Versions 4.3 and 5.1.

The hole is located in the 'login' program that allows people to sign on to the operating system remotely by entering a username and password. The vulnerability can be exploited only if certain remote command protocols, such as Telnet, are enabled, which they usually are by default.

The security hole is serious because of the amount of harm someone could do were they to gain complete control over a vulnerable machine. Once you have super-user access to a machine you can do anything you want – modify files, create them, sniff network traffic, etc.

A fully supported and tested fix is now available from http://sunsolve.sun.com/securitypatch.

AIX news

NetIQ has announced the availability of the first wave of its monitoring and management support for Unix (IBM AIX, Sun Solaris, HP-UX), Linux (Red Hat), and Novell NetWare.

The new cross-platform AppManager modules bring AppManager's Windows-based management and real-time diagnostics capabilities to non-Windows platforms.

In addition, the new cross-platform Extended Management Pack (XMP) modules will extend Microsoft Operations Manager 2000 (MOM) to support centralized monitoring of critical platforms beyond Windows 2000. So it's as easy to manage Unix as it is to manage Windows.

For further information contact:
NetIQ, 3553 N First St, San Jose, CA 95134, USA.
Tel: (408) 856 3000.
URL: http://www.netiq.com/products/am/default.asp.

* * *

IBM has announced multi-year contracts and simplified ordering processes through its new Software Maintenance acquisition model, which includes technical support for IBM distributed software.

Replacing the existing Software Subscription for AIX and OS/400 and stand-alone upgrades previously ordered under each licensed program, the new scheme enables a single site or worldwide enterprise to maintain software entitlement to current or future versions or releases of the eligible products during the Software Maintenance contract period.

For further information contact your local IBM representative.
URL: http://www-1.ibm.com/servers/eserver.

* * *

CommVault Systems has announced support for AIX and leading databases with the latest release of Version 3.7.1 of CommVault Galaxy.

The Galaxy approach offers a unified, scalable data protection solution for distributed, heterogeneous environments, including AIX. With Galaxy, policy-based data protection can be enforced across operating environments and applications, helping users simplify complex storage networks.

For further information contact:.
CommVault Systems, 2 Crescent Place, PO Box 900, Oceanport, New Jersey 07757-0900, USA.
Tel: (732) 870 4000.
URL: http://www.commvault.com/news_story.asp?id=120.

* * *

IBM has introduced entry level p610 Models 6C1 and 6E1, which run both AIX and Linux, and are claimed to eat up 57% less electricity and generate up to 63% less heat than a comparable Sun box, while costing less. A POWER3-II chip running at 333MHz is available.

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