52 AIX update

February 2000

In this issue

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 550955
From USA: 01144 1635 33823
E-mail: harryl@xephon.com

North American office

Xephon/QNA
1301 West Highway 407, Suite 201-405
Lewisville, TX 75077-2150
USA
Telephone: 940 455 7050

Contributions

If you have anything original to say about AIX, or any interesting experience to recount, why not spend an hour or two putting it on paper? The article need not be very long – two or three paragraphs could be sufficient. Not only will you actively be helping the free exchange of information, which benefits all AIX users, but you will also gain professional recognition for your expertise and that of your colleagues, as well as being paid a publication fee – Xephon pays at the rate of £170 ($250) per 1000 words for original material published in AIX Update.

To find out more about contributing an article, see *Notes for contributors* on Xephon's Web site, where you can download *Notes for contributors* in either text form or as an Adobe Acrobat file.

Editor

Harold Lewis

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.

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 1995 issue, are available separately to subscribers for £16.00 ($23.00) each including postage.

AIX Update on-line

Code from *AIX Update* is available from Xephon's Web page at *www.xephon.com/aixupdate* (you'll need the user-id shown on your address label to access it).

Printed in England.

Understanding the more command

The **more** command is an economical way to view text data one screen at a time. The two most common ways of invoking this command are to enter it at the command line followed by a filename and to pipe the output of another command into **more**.

MORE COMMAND BASICS

more displays each screen and waits for a user input. The word ‘more’ appears as the last line of the screen to inform the user that there is more text to view. Pressing the spacebar advances to the next screen of text, while pressing *Enter* advances the text by a line.

Paging forwards through a file is not the only method **more** supports. Once the screens are available and the ‘more’ prompt is being displayed, there are several subcommands that allow **more** to scroll backwards, search for strings, and mark positions for later reference.

The basic syntax of the **more** command is:

```
more flags file
```

where ***flags*** are one or more optional flags and ***file*** is the file that **more** is to process.

FILE EXAMINATION

Suppose you have a large text file called *readme.now*. If you enter:

```
more readme.now
```

the **more** command displays a screenful of text minus two lines used for the display’s status area. The *more* prompt at the bottom indicates the percentage of the file displayed so far.

As mentioned above, you may also pipe output into **more**. Using the example above, entering:

```
more readme.now
```

is the equivalent of entering:

```
cat readme.now | more
```

which writes the contents of the file to the **more** command rather than to the display. But what if the data is not in a file?

Suppose you have a directory that contains hundreds of files, such as */usr/bin*, which typically contains hundreds of executable files. If you enter:

```
ls -l /usr/bin
```

the **ls** command builds a list of files found in */usr/bin* and scrolls the entire list off your display. However, if you enter:

```
ls -l /usr/bin | more
```

the **ls** command pipes the list of files into **more**, which displays the first screen of data, then pauses. At this point, you can enter a subcommand to search for a particular line, to scroll up or down, etc. In contrast with using **more** to display a file, there is no indication of what percentage of output has been displayed. One reason for this is that the volume of data to be output may not be known at this stage.

FLAGS FOR THE MORE COMMAND

Prior to invoking the **more** command, you can enter flags that alter the behaviour of the **more** session, affecting its display and performance and automating tasks.

The list below summarizes flags that can be used with the **more** command.

Subcommand	Description
-d	Displays 'Keys Help'.
-i	Ignores case in search patterns.
-N	Disables line numbering.
-n #	Specifies number of lines per screen.
-p subcmd	Issues specified subcommand.
-s	Compresses multiple blank lines to one blank line.
-v	Disables display of nonprintable characters.
-x tabs	Defines tab stops at the specified position.
-z	Displays backspace, carriage return, and tab characters.

The **-d** flag displays the following message at the bottom of each screen:

```
[Press space to continue, q to quit, h for help]
```

The **-i** flag is used to ignore the case of search patterns specified during the **more** session. The drawback is that you have to specify this flag when invoking **more**.

The **-N** flag is used to disable line numbering. The default behaviour is to track line numbers as files are processed (for instance, to allow line numbers to be passed to **vi**). However, with very large files line number tracking can affect performance. If the **vi** editor is invoked in a session started with the **-N** flag, the **vi** session is initialized at the beginning of the file.

The **-n** flag is used in conjunction with a number to specify the number of lines per screen.

The -p flag is used in conjunction with a **more** subcommand to invoke that subcommand. This is useful if you code a custom **more** command in a script file. For example:

```
more -p 25% projdata.txt
```

displays *projdata.txt*, starting approximately one quarter into the file.

The following example:

```
more -p /Notes projdata.txt
```

displays *projdata.txt* starting at the first occurrence of the word ‘Notes’.

The **-s** flag is used to compress multiple blank lines to one for display. This is useful when viewing text files that are, for instance, formatted for output to a printer.

The **-v** flag is used to disable the display of non-printable characters. Note that the default behaviour of the **more** command is to display non-ASCII and control characters other than the *backspace*, *CR* (carriage return), and *Tab*.

The **-x** flag is used in conjunction with a column setting to define tabs

during a **more** session. The default setting is for a tab stop every eight columns.

The **-z** flag is used to display the *backspace*, *CR*, and *tab* control characters (as mentioned above, the default is *not* to display these control characters).

SUBCOMMANDS

Once **more** displays the desired file, subcommands can be used to move about, edit the file, and run AIX commands. Some subcommands are set up to mimic **vi** editor commands, which helps you remember them if you're familiar with **vi**. The focus line is the line containing the target data, which is typically displayed on the third line of the screen.

Although this section discusses subcommands that are available when **more** is invoked with a file name, most also apply when **more** displays output piped from another command.

The list below summarizes subcommands that can be used with the **more** command. Note that you substitute a number for the **#** sign.

- **q**, **Q**, or **:q**

  Quits from the **more** session.

- **f**, 'spacebar', or ***Ctrl***+**F**

  Scrolls forward one screen.

- **b** or ***Ctrl***+**B**

  Scrolls back one screen.

- **d**

  Scrolls down half a screen.

- **#d** or ***#Ctrl***+**D**

  Sets half- screen line value and scrolls down that many lines.

- **u**

  Scrolls up half a screen.

- **#u** or ***#Ctrl*+U**

  Sets the half-screen line value and scrolls up that many lines.
- **j**

  Scrolls forward one line.
- **#j**, **#**, or **#Ctrl+E**

  Scrolls forward a specified number of lines.
- **k**

  Scrolls back one line.
- **#k** or ***#Ctrl*+Y**

  Scrolls back a specified number of lines.
- **g**

  Moves back to the first line in the file.
- **#g**

  Moves to the specified line number.
- **G**

  Moves forward to the last line in the file.
- **#G**

  Moves to the specified line number.
- **#p** or **#%**

  Moves the specified percentage into the file.
- **/*pattern***

  Searches forward for a specified pattern.
- **?*pattern***

  Searches backward for a specified pattern.
- **/!*pattern***

Searches forward for the first line not containing the specified pattern.

- **?!*pattern***

  Searches backward for the first line not containing the specified pattern.

- **n**

  Performs the next search using the previous search pattern and search direction.

- **ma-z**

  Marks a position for later reference.

- **'a-z**

  Moves to a specified reference point.

- '' (two single quotes)

  Returns to the starting point after a move to a marked position, full file, percentage, or a search operation.

- **!*command*** or :**!*command***

  Runs the specified AIX command.

- **h**

  Displays the **more** command help screen.

- **v**

  Edits the current file using **vi** editor.

- **r** or ***Ctrl***+**L**

  Refreshes the display.

q quits the **more** session. You can also quit with **Q** or **:q**.

Use **f** and **b** (or ***Ctrl***+**F** and ***Ctrl***+**B**) to scroll the display forward or backward one screen. Since the most common use of the **more** command is to page forward through a file, the spacebar has also been set up to advance the display forward one screen.

Use the **d** and **u** (or ***Ctrl***+**D** and ***Ctrl***+**U**) to scroll the display down or up half a screen.

If you precede the **d** or **u** subcommand with a number, **more** sets the 'half page value' to the number of lines you specified (the setting persists for the duration of the session). So, if you enter **7d**, **more** scrolls down seven lines, and seven lines again whenever **d** is pressed in that session. This also causes **more** to scroll up seven lines whenever **u** is pressed.

This type of scrolling is useful when you wish to view data in increments other than full screens, particularly 'stanza'-like data, which has a fixed periodicity in terms of rows.

Use **j** and **k** to scroll through the file line by line. **j** scrolls forward a line and **k** scrolls back a line. The *Enter* key has also been set up to scroll forward one line. You can also use ***Ctrl***+**Y** and the ***Ctrl***+**E** to move up and down through the file line by line.

If you precede the command with a number, **more** will scroll that many lines.

The command **G** is used to move forward to the last line of the file, and **g** to move back to the first line. If you precede the command with a number, **more** scrolls up or down to that line number.

You can also move through a file percentage-wise using the commands **p** and **%**. For example, if you issue the command **50p** or **50%** anywhere in a file, **more** takes you approximately to the midpoint of the file. If you were on the last screen and typed **10%**, **more** would take you to the screen that's approximately one-tenth into the file from the top. This can be useful if you want to move quickly through a large file without regard to line numbers.

Notes:

1 The specified percentage moves the focus to that position from the top of the file, not from your current location in the file.

2 All percentages used by **more** are calculated using the number of characters, not the number of lines. For most data, individual lines probably contain approximately equal numbers of characters.

However, if you have a file with 60 lines of 132 characters followed by 300 lines of eight characters, the file is 'top heavy' and would skew the percentages towards the top of the file.

3 The percentage that appears with the *more* prompt represents the amount of the file that has been displayed so far, including the current screen. This may differ from the percentage you specified with the **p** or **%** subcommand, which moves the focus, which is a line near the top of the screen, to the line that divides the file by the percentage specified. The discrepancy is particularly noticeable when smaller files are displayed.

SEARCHES

Searching forwards and backwards using a pattern

Use a forward slash ('/') followed by a search pattern to search the file forward from the current focus. Use the question mark ('?') and a search pattern to search for text appearing before the current focus.

For example, if your current focus is approximately mid-way through the file, and you enter the command sequence:

```
/VARIABLE
```

more positions the focus at the first occurrence of 'VARIABLE' found below your starting point.

If you enter:

```
?VARIABLE
```

more finds the first occurrence of 'VARIABLE' before your starting point.

Searching forwards and backwards for no occurrence of pattern

If you precede the search pattern with an exclamation mark ('!'), **more** looks for the first line that doesn't contain a string that matches the specified search pattern. Specifying **/!*pattern*** searches forwards and **?!*pattern*** searches backwards. For example, if you were to enter:

```
/!INCLUDE
```

more skips several consecutive lines that contain the string ‘INCLUDE’ and displays the first line that doesn’t.

Repeat search

Use the **n** key to tell **more** to position the focus at the next occurrence of the specified search string. If the direction of the last search was backwards, the **n** key causes the search to be repeated in the same direction.

ADVANCED MOVEMENT

Marking a screen

Use the **m** key followed by a character from ‘a’ to ‘z’ to mark a position for later reference. Then use the ‘'’ (single quote) key followed by the desired character to cause **more** to position the line of focus at that reference point.

Type ‘'’ (two single quotes, not a double quote) to return to the focus line prior to the move to the marked line (this works only if the movement is greater than a page).

For example, if the current focus line were to display a screen containing a conversion table or ‘key’, you could type **mk** and **more** will mark the position as ‘k’. Later, while viewing a screen containing a data item for which you would like to see the conversion, type **'k** (single quote k) and **more** will redisplay the screen containing the key. To return to the text, type **''** (two single quotes).

The use of two single quotes is not restricted to moving to a marked line, and they can also be used to return after movements using the commands **g** or **G**, percentages **P** or **%**, and searches.

MISCELLANEOUS SUBCOMMANDS

Start an AIX command

Use the exclamation mark (‘!’) followed by an AIX command to open a new shell and run the specified command. You can also use the **:!*cmd*** combination (colon, exclamation mark, command) if desired.

For example, if you would like to initiate a background process called **mytask** during a **more** session, enter **!mytask&** at the **more** prompt. You will see a message that the command is executed.

Display the more command help screen

Use the **h** key to display help screens for the **more** command. Use the spacebar to view successive screens and **q** followed by the *Enter* key to return to the **more** session.

Start the vi editor

Use the **v** key to begin editing the current file using the **vi** editor. The default behaviour of the **more** command is to position the editor cursor at the same line as **more**'s focus when **v** is typed. If you want the editor to initialize with line number 1 being displayed each time **vi** is invoked, use the **-N** flag when invoking the **more** command.

Refresh the display

Use the **r** key to refresh the display. This is useful when an external process is making changes to the data currently displayed. You can also use ***Ctrl***+**L** to perform this operation.

SUMMARY

There's more to the **more** command than just scrolling forward through a file. Once you practise using **more**, you will probably find yourself using the command frequently. It is a good way to inspect a file without the implicit dangers of editing it. It is also one of the few ways to inspect the output of certain commands without overtaxing disk resources.

David Chakmakian
Programmer (USA)

A collection of utilities

Unix has many fascinating small utilities that do one or more simple jobs. Many are designed to work in conjunction with other programs. Most of these commands work well as part of a pipe, and are frequently combined with other commands in order to get some job done.

The commands are too simple for each to warrant a whole article, but they are also far too useful to ignore. This article will cover several of these ‘small fry’ commands.

TAIL

The **tail** command displays the last ten lines of a file on standard output (usually the terminal). The number can be modified by using a ***–nn*** switch, where ***nn*** is the number of lines to display. For example:

```
tail -20 log.txt
```

displays the last 20 lines of *log.txt*. If a plus sign (‘+’) is used instead of a dash (‘-’), the number of lines specified is understood to be from the beginning of the file. For example:

```
tail +20 log.txt
```

skips lines 1 to 19 of *log.txt* and display all lines from the twentieth to the end of the file.

tail is particularly useful for looking at the last entries in a log file. To this end, it has a very useful **-f** option. If **tail** is used with this flag, the file being displayed is not closed, but is instead kept open. The **tail** program sleeps for one second, then wakes and checks whether new lines were added to the file. If they were, the new lines are displayed. This option is useful for monitoring a log file that is actively being written to. Assuming that *log.txt* is a log file in use by one or more programs, then:

```
tail -f log.txt
```

displays the last 10 lines of *log.txt*, and then updates the screen each

second with any new records that are added to *log.txt*. I first used **tail -f** to monitor a file that was being filled with transactions by cash registers as cashiers rang up purchases. **tail** really is an excellent debugging tool. For testing purposes, we set up the register, rang up different types of transaction, and then examined the result on the Unix screen, immediately. In this way we were quickly able to isolate transactions that were being written to the file incorrectly.

Because **tail** routes its output to standard output, it is possible to pipe the results into another process. At one point when debugging the registers, we were concerned that they were writing garbage characters and/or nulls to the transaction file. The **tail** utility processes files as text, so it's not possible to see nulls and other non-printable characters in the data. We used **tail** to pump the data to **od** ('octal display') and then display the data in hex, so that we could examine it for null characters. If **tail -f** is used at the beginning of the pipe, the transaction remains open and constantly pumps data to **od**, where the bytes are translated into hex and displayed.

```
tail -f trx.txt|od -xc
```

You may want to try the following example to test **tail**. Move to a directory where you can create some files without interfering with any production data. Next create a process that writes to a log file. The Korn shell is AIX's default shell – if you are not already running it, start it by typing **ksh** and pressing 'Enter'. Type the lines in Listing 1 below. As you enter each line, the '>' prompt appears, indicating that more input is expected. The lines in Listing 1 are numbered to allow them to be identified easily, though the numbers should not appear on the screen. The initial '$' on line 1 and the '>' on lines 2 to 6 are provided as prompts for your input – you type everything to the right of the first character. The command creates a process that sleeps for two seconds and then awakens and appends the date and time to *log.txt*. The process is submitted in the background by enclosing all of the commands in parentheses in lines 1 and 6, and ending with a final ampersand ('&') in line 6. When you press *Enter* at the end of line 6, the process is submitted in the background and runs as a detached job with no terminal to write to. It doesn't need a terminal since it's writing to *log.txt*. On line 7, the operating system responds by giving you the job number of the job that is now running in the background. In the

example, it's '657'. Make a note of the job number that appears on your terminal, as you will need it later to kill the process.

LISTING 1: PUMPING DATA INTO A LOG FILE

```
1    $ ( while true
2    > do
3    > sleep 2
4    > echo `date` >>log.txt
5    > done
6    > )&
7    [1]   657
8    $
```

Now that *log.txt* is being filled with a date and time stamp every two seconds, enter the following command:

```
tail -f log.txt
```

As you watch the screen, *log.txt* is gradually filled with information, and **tail -f** continues to display the information on the screen as in Listing 2.

LISTING 2: SAMPLE OUTPUT FOR TAIL -F

```
$ tail -f log.txt
Sun Apr 12 14:03:01 PDT 1998
Sun Apr 12 14:03:03 PDT 1998
Sun Apr 12 14:03:05 PDT 1998
Sun Apr 12 14:03:07 PDT 1998
```

Press *Control-C* or the *Delete* key to stop your **tail -f** process, depending on how your terminal is set up.

Finally you need to stop the background process that is logging lines to the *log.txt* file. Using the job number that you noted after the process started, type:

```
kill 657
```

The system will usually respond with a message that the specified job has been terminated, as in the following example (the continuation character, '➤', indicates that one line of code maps to more than one line of print):

```
$ kill 657
```

```
[1] + Terminated     (while true;do;sleep 2;echo `date`
➤  >>log.txt;done)&
$
```

DATE

While **date** is commonly used to set the date and time, it can also be used to extract the date and time in numerous formats.

The simplest form of the command is the default one:

```
$ date
Sun Apr 12 14:03:01 PDT 1998
```

A simple logging script could be created using this format as in Listing 3. After you have edited this with **vi**, save it as 'logit.sh'. Then change its permissions by typing **chmod a+x logit.sh** to make the script executable.

LISTING 3: LOGIT.SH

```
# logit.sh
# log all the arguments passed on the command line along
# with the date and time stamp
echo `date` $@ >> log.txt
```

Run **logit.sh** by typing:

```
$ logit.sh This is a line to log
```

If you then use **cat** to type the contents of *log.txt*, the file should contain:

```
$ cat log.txt
Sun Apr 12 14:03:01 PDT 1998 This is a line to log
```

The date may also be formatted by using a '+' followed by formatting characters, as in:

```
$ date +%D
04/12/98
```

%D is a date formatting character that indicates that the date is to be output in *mm/dd/yy* format. Some of the other formatting characters are listed below. Your version of **date** may not support all of them, and may support others not listed below. Check your **man** pages for a list of formatting characters.

%A The full weekday name (Sunday, Monday, etc).

%a The abbreviated weekday name (Sun, Mon, etc).

%B The full month name (January, February, etc).

%b The abbreviated month name (Jan, Feb, etc).

%C The first two digits of the year (00 to 99).

%D The date in the form *mm/dd/yy*.

%d The two-digit day of the month (01 to 31).

%H The hour in 24-hour format (00 to 23).

%I The hour in 12-hour format (01 to 12).

%j The numeric day of the year (001 to 366).

%M The minute (00 to 59).

%m The month number (01 to 12).

%n A ‘newline’ character.

%p The equivalent of ‘AM’ or ‘PM’.

%R The 24-hour time (for example, 13:22).

%r The 12-hour time with AM/PM (for example, 11:53:29 AM).

%S The second (00 to 59).

%T The 24-hour time (14:53:29).

%U The week number in the year (00-53), where Sunday is the first day of the week and all days before the first Sunday of a year are in week 0.

%u The weekday number (Monday = 1, Sunday = 7).

%w The weekday number (Sunday = 0, Saturday = 6).

%Y The year in four-digit format (1999).

%y The year in two-digit format (99).

%Z The time zone name (PDT, EDT, etc).

It is often necessary to include spaces in a date format, so it is common to include the date format string in quotes. In the following example, the quotes are necessary because of the space in the format string.

```
$ date "+%D %R"
04/12/98 13:22
```

The ability to format date fields can be used to create log file names that contain a date stamp.

In Listing 4, a new version of **logit.sh**, creates a log file name using the year, month, day, and hour. This ensures that log files don't become too large to be handled easily.

LISTING 4: LOGGING NAMES WITH A DATE STAMP

```
# logit.sh
# log all the arguments passed on the command line, in an
# hourly log file.
logtime=`date "+%y%m%d%H"`
logfile=${logtime}.log
echo `date` $@ >> $logfile
```

This could include the minute if sufficient events are logged to warrant it, in which case use the code in Listing 5.

LISTING 5: LOG FILE NAMES THAT INCLUDE THE MINUTE

```
# logit.sh
# log all the arguments passed on the command line.
logtime=`date "+%y%m%d%H%M"`
logfile=${logtime}.log
echo `date` $@ >> $logfile
```

One advantage of using the date to create log file names is the natural sort order provided by the **ls** command. If the file name is created by concatenating the year, month, day, and hour (or hour and minute), the log files are sorted by time of creation when an **ls** command is used.

READ

The **read** command is built into the Korn shell. Its purpose is to allow input information to be read from standard input (usually the keyboard).

Type the following command, then enter a single word (with no spaces) as in the example below. The command **read x** causes input to be read from standard input and assigned to the variable *$x*, which is then echoed to the screen.

```
$ read x; echo $x
hello
hello
$
```

read is used in shell scripts to accept user input and assign it to a variable. Listing 6, **simpmenu**, is a two-option menu that uses **read** to accept the user's input. Line numbers are included for convenience, though they are not part of the script.

A menu is generated in lines 7 to 18. After line 19 reads the user selection into variable *x*, one of two possible actions is executed at lines 20 to 23 and 25 to 28. This repeats until the user enters a '9' as the menu selection. This is controlled by line 5, which states that if *$x* is not equal (*-ne*) to '9', the loop continues to execute.

Note the additional **read**s in lines 22 and 27 – while they seem to have no variable nominated for input, the **read** command uses the default variable *REPLY* if no variable is named. Lines 22 and 27 give the user the opportunity to see the last page of output before the screen is cleared.

LISTING 6: SIMPMENU USING 'READ'

```
 1   # simpmenu
 2   # a simple menu program
 3
 4   x=1
 5   while [ $x -ne 9 ]
 6   do
 7       clear
 8       echo
 9       echo
10       echo "Enter your selection"
11       echo
12       echo
13       echo "1  Display directory"
14       echo "2  Display processes"
15       echo
16       echo
```

```
17        echo "9  Exit"
18
19        read x
20        if [ $x -eq 1 ]
21        then
22            ls -l|more ; echo "Press Enter" ; read
23        fi
24
25        if [ $x -eq 2 ]
26        then
27            ps -ef|more ; echo "Press Enter" ; read
28        fi
29    done
```

Using the default variable *REPLY*, the first exercise could be shortened to:

```
$ read; echo $REPLY
hello
hello
$
```

The **read** command can be used to read and fill more than one variable. Try entering the following command and then typing four words separated by spaces.

```
$ read x y z; echo $x; echo $y; echo $z
one two three four
one
two
three four
$
```

The **read** command reads in all of the words on a line and assigns them one by one to the variables listed after the **read** command. If there are more words than there are variables, the remaining words are all assigned to the last variable. In the above example, *$x*=‘*one*’, *$y*=‘*two*’, and *$z*=‘*three four*’.

This is true even with only one variable, as in the example below. The *$x* variable is the last variable associated with **read**, and so it takes the value ‘one two three four’.

```
$ read x; echo $x
one two three four
one two three four
$
```

You can break up the list in *$x* by using a *for* statement, as in the example below.

```
$ read x; for name in $x; do echo $name; done
one two three four
one
two
three
four
$
```

Listing 7, **multmenu**, shows a script that allows a user to enter several menu selections separated by spaces. Lines 20 to 34 break the user selection into parts, each being processed in multiple passes.

LISTING 7 MULTMENU ALLOWS MULTIPLE MENU PICKS

```
 1   # multmenu
 2   # a multiple menu program
 3
 4   x=1
 5   while [ $x -ne 9 ]
 6   do
 7       clear
 8       echo
 9       echo
10       echo "Enter your selection"
11       echo
12       echo
13       echo "1  Display directory"
14       echo "2  Display processes"
15       echo
16       echo
17       echo "9  Exit"
18       read x
19       for pick in $x
20       do
21          if [ $pick -eq 1 ]
22          then
23              ls -l|more ; echo "Press Enter" ; read
24          fi
25          if [ $pick -eq 2 ]
26          then
27              ps -ef|more ; echo "Press Enter" ; read
28          fi
29          if [ $pick -eq 9 ]
30          then
31              exit
```

```
32          fi
33       done
34   done
```

There is one other powerful trick that works with **read**. It is possible to start a process running that produces output, and then read that output as if it were coming from the keyboard. This requires two steps:

1 The process has to be started in a special way

2 The **read** command has to be informed that it is reading from another process rather than the keyboard.

Listing 8, **oldest.sh**, uses this technique. The **ls -tr** command lists the files in a directory in chronological order, starting with the oldest. Line 4 in the script launches the **ls -tr** command using a pipe bar and an ampersand '|&'. The ampersand indicates that the command is to be detached and run in the background. The pipe indicates that a pipeline is to be created between the **oldest.sh** process and the detached process. Line 6 issues a **read** with the **-p** flag, meaning that it is to read from a pipe rather than standard input. **read** reads everything up to the first newline character, thus reading the first line output by **ls -tr**.

LISTING 8: READING FROM ANOTHER PROCESS

```
1   # oldest.sh
2   # names the oldest file
3
4   ls -tr|&
5
6   read -p x
7
8   echo "The oldest file is" $x
```

Another example of using **read** appears in the next section on **wc**. As you'll see, **read** is useful for more than just simple input.

WC

The **wc** command provides a count of words, lines, and characters in a document. In Listing 9, *log.txt* contains 154 lines, 918 words, and 4431 characters. **ls -l** reveals the size of the file to be 4431 bytes, which matches the character count produced by **wc**.

LISTING 9: USING WC

```
$ wc log.txt
  154  918  4431 log.txt
$ ls -l log.txt
-rw-r--r--   1 mjb  group      4431 Apr 12 14:35  log.txt
```

The **wc** word counter defines a word as any non-blank sequence of letters or numbers.

You may limit the output of **wc** to a count of only words, lines, or characters using the **-w**, **-l**, and **-c** switches respectively. The default is to display all three values. The display output of **wc** includes the file name so you can output the results for more than one file by using wild cards on the command line. The following example displays the line counts for all files that start with 'log', along with a total.

```
$ wc -l log*
  154  log.txt
    5  logit.sh
  159  total
```

The **wc** utility is useful for authors. Listing 10, **ueditor**, is my own personal editor. It doesn't care about the quality of the content, but it does provide prods and praise until I have produced 2000 words. It includes line numbers for explanation. The script is invoked using the command **ueditor article.txt**. Line 3 shows **wc** being used to count words. The terminator '|&' causes the **wc** command to start as a background task and opens a pipe from the background task back to **ueditor** in such a way that the output of the task can be read as input by **ueditor**. The output is read in line 4. The command **wc -w** outputs two fields: the word count and the file name. These are read into separate variables, *$wrds* and *$name*, in line 4. The value in *$wrds* is tested in line 5 – if it's less than (**-lt**) 2000, words of encouragement are printed. If *$wrds* is 2000 or more, the message contains some praise for the humble and hard-working author.

LISTING 10: UEDITOR

```
1   # ueditor
2   # checks if the article has 2000 words yet
3   wc -w $1|&
4   read -p $wrds $name
5   if [ $wrds -lt 2000 ]
```

```
6   then
7       echo "Only" $wrds "words! More, more I'm still not satisfied."
8   else
9       echo $name " is a brilliant piece!"
10   fi
```

DU

If the directory only files contains, it is easy enough to issue the command **ls -ls** to get the size of files in bytes and blocks:

```
$ ls -ls
total 6
   2   -rw-r--r-- 1 mjb   group     3 Feb 04 23:31 minutes.txt
   4   -rw-r--r-- 1 mjb   group  1201 Feb 04 23:25 note.txt
```

The first column contains the size of the file in 512-byte blocks, and the sixth column gives the size in bytes. Files in this directory consume six blocks that contain only 1204 bytes.

This method of allocating chunks of disk space to a file, even if the file doesn't use all the space, is used in all major operating systems in one form or another. Some convenient number of bytes is selected as the minimum that can be allocated to a file. This is an *allocation unit*. If the file does not use all the space in an allocation unit, what there is of the file is recorded at the beginning of the unit and the remainder of the unit is set aside for further expansion of the file. If the file is expanded, as long as it doesn't exceed the number of bytes in an allocation unit, all the new bytes are stored in the empty reserved space on the disk. Once the file size exceeds its initial allocation, another allocation unit is grabbed and reserved for the file. The spillover from the first allocation unit is tucked at the start of the second allocation unit and so on. Earlier Unix systems used an allocation unit of 512 bytes. These 512 bytes came to be known as a block. As disk sizes grew, the basic allocation unit was increased to 1024 bytes on most systems (larger on some), but many utilities, such as **ls**, still report file size or disk use in 512-byte blocks. So the three-byte file uses two blocks.

In the following example, the directory in question includes a subdirectory, *perl*. The two blocks allocated to the *perl* directory are only the blocks used by the directory entry itself, not ones used by files contained in the directory.

```
$ ls -ls
total 6
   2 -rw-r--r--  1 mjb   group      3 Feb 04 23:31 minutes.txt
   4 -rw-r--r--  1 mjb   group   1201 Feb 04 23:25 note.txt
   2 drwxr-xrx   2 mjb   group    128 Jan 29 18:53 perl
```

We could figure out the sizes by doing an **ls -ls perl**, but suppose there is another directory under *perl*, and so on?

The answer to this dilemma is the Unix utility **du**. How do you **du**? The **du** utility analyses directories and subdirectories recursively and displays blocks used. The display below provides information that the directory being processed contains a *perl* subdirectory. This, in turn, contains a subdirectory, *src*. The *src* directory contains files totalling 1540 blocks. The *perl* directory count includes all the blocks in *src* plus the blocks used by files in *perl*. Finally, the top-level includes all blocks below it plus blocks used by files used in the current directory.

```
$ du
1540 ./perl/src
5648 ./perl
5654 .
```

Using the **-a** option causes the detail of each file to be displayed.

```
$ du -a
1500 ./perl/src/big.prl
40   ./perl/src/prog.prl
1540 ./perl/src
4108 ./perl/perl.tar
5648 ./perl
2    ./minutes.txt
4    ./note.txt
5654 .
```

The **du** command cuts through a lot of what **ls** commands do and provides size information plus a reasonable display of the directory tree.

TR

The **tr** utility translates one set of characters into another set.

The command:

```
tr abc def <test.txt
```

processes the records from *test.txt* and translates *a* to *d*, *b* to *e*, and *c* to *f*. At first glance this doesn't seem very useful unless you want to practise very amateur cryptography; however, **tr** has additional options that make it much more useful. Two examples should suffice to give you a feel for the command. The characters to be translated can be expressed as a range. In the first example below, a directory is output through **tr**, which translates *a* to *A*, *b* to *B*, and so on, converting everything to uppercase.

```
$ ls -ls|tr [a-z] [A-Z]
TOTAL 6
   2 -RW-R--R--   1 MJB   GROUP      3 FEB 04 23:31 MINUTES.TXT
   4 -RW-R--R--   1 MJB   GROUP   1201 FEB 04 23:25 NOTE.TXT
   2 DRWXR-XRX    2 MJB   GROUP    128 JAN 29 18:53 PERL
```

One use for case conversion is to work around a problem caused by some utilities that copy MS-DOS files. They copy the files using MS-DOS's uppercase convention, and the file names need to be converted to lowercase to follow Unix's naming style and work correctly. The command below changes all file names to lowercase. It takes each file name and echoes it through a pipe using **tr** to change the case. The result is used as the target of an **mv** command.

```
$ for name in *
> do
> mv $name `echo $name|tr [A-Z] [a-z]`
> done
$
```

tr's **-s** switch causes repeated instances of a character to be translated to just one instance. In the following example, the file *test.txt* contains a line with multiple spaces between words. While the **tr** command seems to do nothing – after all, it just translates a space to a space – the **-s** switch squeezes multiple spaces into a single output space. The resulting **test2.txt** has single spaces between words.

```
$ cat test.txt
How     are   you           today?
$ tr -s " " " " < test.txt >test2.txt
$ type test2.txt
How are you today?
```

Mo Budlong (USA)

SSCCARS (part 4)

This is the penultimate part of this article on a group of utilities that together comprise a source code management system. The first part of this article appeared in the November 1999 issue of *AIX Update* (Issue 49) and the final part appears in next month's issue (Issue 53).

EXECSU.C

execsu is a C module that performs tasks that require root privilege, such as copying a source file to or from a directory that may only be written to by root.

Note the use of the continuation character, '➤', in the code to indicate that one line of code maps to more than one line of print.

EXECSU.C

```
/*********************************************************************
* Author      : Arif  Zaman
*
* Name        : execsu.c
*
* Description : Perform tasks that require superuser privilege.
*
* Arguments   : 1 An action type, the ones available being:
*                 CR = Create a lock file in lock directory.
*                 RM = Remove a file from any source-related
*                      directory.
*                 WL = Write a log entry.
*                 UL = Update a log.
*                 CI = Copy the source being checked in.
*                 RS = Register the source.
*
*               2 A file name.
*               3 A directory name.
*               4 The date for log message.
*               5 The time for log message.
*               6 The user-id for the log message.
*               7 The module name being checked in or out.
*               8 The log message.
*
* Notes       : 1 The program is owned by root and has its uid
*                 bit set. Any process executing it effectively
```

```
*                  becomes root, the owner of the program.
*
*               2 The uid bit can be set for the shell script but
*                  the shell ignores it completely; it has to be a
*                  compiled program.
*
*               3 The program is used mainly in conjunction with the
*                  various in-house source code control utilities.
*
*               4 The number of command line arguments varies
*                  according to the type of action required.
*****************************************************************/
#include <stdio.h>
/*
 * FUNCTION PROTOTYPE
 */
short   main (int argc, char *argv[]);
short   ParseCommandLine (int arg_count, char **argval);
short   ProcessAction (void);
short   ProcessSourceDetails (void);
short   WriteSourceDetails (void);
short   WriteLog (void);
short   CheckWordSplit (char *line);
char *  StripLeadingSpaces (char *string);
/*
 * MODULE CONSTANTS
 */
#define  SUCCESS   0
#define  FAILURE   1
#define  USAGE "Usage: execsu <action> <file name> <directory name>
➤  <checkin/out date> < check in/out time> <check in/out userid>
➤  <module name> <comment>\n"
#define TRUE  1
#define FALSE 0
/*
 * GLOBAL VARIABLES
 * variables for manipulating lock and log files
 */
char   action_type[3];          /* mandatory command line arguments */
char   file_name[60];
char   dir_name [60];
char   chk_in_out_date[15];     /* used for action type WL and UL   */
char   chk_in_out_time[20];
char   chk_in_out_uid[30];
char   chk_in_out_module[30];
char   chk_in_out_comment[100];
char   source_full_name[31];     /* used for action type RS         */
char   source_short_name[21];    /* used for action type RS         */
char   source_description[2001];
char   source_details[81];       /* source details record           */
```

```
short  total_char_processed ;    /* number of character processed   */
char   source_name_file[100];    /* file for source details         */
short  use_hypen;

/*********************************************************************
* Name        : main
*
* Description : Calls all other functions.
*
* Returns     : SUCCESS or FAILURE
*********************************************************************/
short main (int argc, char *argv[ ])
{
  if (ParseCommandLine (argc,argv) != SUCCESS)
    return FAILURE;
  if (ProcessAction () != SUCCESS)
    return FAILURE;
  return SUCCESS;
}
/*********************************************************************
* Name        : ParseCommandLine
*
* Description : The function parses the command line.
*
* Returns     : SUCCESS or FAILURE
*********************************************************************/
short ParseCommandLine(int arg_count, char **argval)
{
  /*
   * check argument count
   */
  if (arg_count > 9)
  {
    printf(USAGE);
    return FAILURE;
  }
  /*
   * copy arguments
   */
  strcpy (action_type,*(argval+1));
  strcpy (file_name,*(argval+2));
  strcpy (dir_name,*(argval+3));
  if ((strcmp (action_type,"WL") == 0) ||
      (strcmp (action_type,"UL") == 0))
  /*
   * copy additional arguments
   */
  {
    strcpy (chk_in_out_date,*(argval+4));
    strcpy (chk_in_out_time,*(argval+5));
```

```
    strcpy (chk_in_out_uid,*(argval+6));
    strcpy (chk_in_out_module,*(argval+7));
    strcpy (chk_in_out_comment,*(argval+8));
  }
  else if (strcmp (action_type,"RS") == 0)
  /*
   * copy additional arguments
   */
  {
    strcpy (source_short_name,*(argval+4));
    strcpy (source_full_name,*(argval+5));
    strcpy (source_description,*(argval+6));
  }
  return SUCCESS;
}
/*********************************************************************
* Name        : ProcessAction
*
* Description : Acts according to the action types below.
*               CR - Create a lock file in the target directory.
*               RM - Remove a file from a directory.
*               WL - Write a log message in a specified file in
*                    target directory.
*               UL - Update log file.
*               CI - Check in a source.
*
* Returns     : SUCCESS or FAILURE
*********************************************************************/
short   ProcessAction (void)
{
  FILE  *fptr;                          /* general file pointer      */
  short  rc;                            /* return code               */
  short  i;
  char   lock_file[100];                /* used for CR action        */
  char   source_file[100];              /* used for action type = CI */
  char   target_file[100];
  char   system_command[100];           /* shell command             */
  if (strcmp(action_type,"CR") == 0)
  {
    /*
     * create a lock file and generate lock file name
     */
    strcpy(lock_file,dir_name);
    strcat(lock_file,"/");
    strcat(lock_file,file_name);
    /*
     * open and close the file
     * for creating a zero byte file
     */
    if ((fptr = fopen(lock_file,"w")) == NULL)
```

```
    {
      printf("execsu:ERROR: Failed to create the file %s\n",lock_file);
      return FAILURE;
    }
    fclose(fptr);
    return SUCCESS;
  }
  else if (strcmp(action_type,"RM") == 0)
  {
    /*
     * remove a file from a  directory
     */
    strcpy(system_command," rm ");
    strcat(system_command,dir_name);
    strcat(system_command,"/");
    strcat(system_command,file_name);
    rc = system(system_command);
    if (rc == SUCCESS)
      return SUCCESS;
    else
    {
      printf("execsu:ERROR: Failed to remove file %s in %s
             ➤  directory\n", file_name,dir_name);
      return FAILURE;
    }
  }
  else if (strcmp (action_type ,"CI") == 0)
  {
    /*
     * check in the source
     */
    strcpy(target_file,dir_name);
    strcpy(source_file,file_name);
    strcpy(system_command,"mv ");
    strcat(system_command,source_file);
    strcat(system_command," ");
    strcat(system_command,target_file);
    rc = system(system_command);
    if (rc == SUCCESS)
    {
      /*
       * change the ownership to root
       * change the group to system
       * change the permission to 744
       */
      strcpy(system_command,"chown root ");
      strcat(system_command,target_file);
      system(system_command);
      strcpy(system_command,"chgrp system ");
      strcat(system_command,target_file);
```

```
      system(system_command);
      strcpy(system_command,"chmod 744  ");
      strcat(system_command,target_file);
      system(system_command);
      return SUCCESS;
    }
    else
    {
      printf("execsu:ERROR: Failed to move the source file %s\n",
             file_name);
      return FAILURE;
    }
  }
  else if (strcmp(action_type,"RS") == 0)
  {
    /*
     * write source details in the file
     */
    if (ProcessSourceDetails () !=  SUCCESS)
      return FAILURE;
  }
  else if ((strcmp(action_type,"WL") == 0) ||
           (strcmp(action_type,"UL") == 0))
  {
    if (WriteLog() != SUCCESS)
    return FAILURE;
  }
  return SUCCESS;
}
/*********************************************************************
* Name        : ProcessSourceDetails
*
* Description : The function writes the source details to the
*               specified data file.
*
* Returns     : SUCCESS
*
* Notes       : 1 The maximum length of a source detail record is
*                 80 and it has the following layout:
*                 File Name     Full Name     Description
*                     22           32             26
*                                                 26
*                                                 26
*                                                 22
*               2 The number of records that have to be written
*                 for source details depends on the length of
*                 source description, the maximum being 100.
*********************************************************************/
short ProcessSourceDetails(void)
{
```

```
short  source_desc_len;
short  numchars;        /* characters in multi-record description */
char  *curloc;          /* location of source description         */
short  char_remaining ; /* characters still to be processed       */
short  i;
char   formatted_source_desc[2001]; /* for description formatting */
char  *word_location ;
char   string[2001] ;
/*
 * format source description (a white space is a word delimiter)
 */
word_location = source_description;
memset(formatted_source_desc,'\0',2001);
while (TRUE)
{
  memset(string,'\0',2001);
  sscanf(word_location,"%s",string);
  if (! strlen(string))
    break ;
  strcat(formatted_source_desc,string);
  strcat(formatted_source_desc," ");
  word_location=word_location + strlen(string) + 1 ;
}
/*
 * prepare source registration data file name
 */
strcpy(source_name_file,dir_name);
strcat(source_name_file,"/");
strcat(source_name_file,file_name);
/*
 * prepare record with source details
 */
memset(source_details,'\0',81);
strcpy(source_details,source_short_name);
/*
 * right pad the string to 20 (file name) + 2 spaces
 */
for (i = strlen(source_details);i <22 ; i++)
  source_details[i] = ' ';
source_details[i+1] = '\0';
/*
 * append source full name (30 characters + 2 spaces)
 */
strcat(source_details,source_full_name);
/*
 *  left pad the record for first 54 characters
 */
for (i = strlen(source_details) ; i < 54; i++)
  source_details[i] = ' ';
source_details[i+1] = '\0';
```

```
source_desc_len = strlen(formatted_source_desc);
total_char_processed = 0;
char_remaining = source_desc_len ;
curloc=formatted_source_desc;
numchars=source_desc_len ;
while (TRUE)
{
  use_hypen = 0;
  if (char_remaining < 27)
    break ;
  /*
   * curloc = StripLeadingSpaces(
   *               &formatted_source_desc[total_char_processed]);
   */
  numchars = CheckWordSplit(
                  &formatted_source_desc[total_char_processed]);
  /*
   * numchars=CheckWordSplit(curloc);
   */
  strncat(source_details,
          &formatted_source_desc[total_char_processed],numchars);
  total_char_processed = total_char_processed + numchars;
  if (use_hypen == 1)
    strncat(source_details,"-");
  /*
   * right pad the description to its entire length
   *   for (i = strlen(source_details) ; i < 80; i++)
   *     source_details[i] = ' ';
   *   source_details[i+1] = '\0';
   */
  strncat(source_details,"\n");
  /*
   *  write this record
   */
  if (WriteSourceDetails () != SUCCESS)
    return FAILURE;
  memset(source_details,'\0',80);
  /*
   * left pad the new record up to 54 characters
   */
  for (i = strlen(source_details) ; i < 54; i++)
    source_details[i] = ' ';
  source_details[i+1] = '\0';
  char_remaining = source_desc_len - total_char_processed ;
}
/*
 * append the description record
 */
curloc = StripLeadingSpaces(
                   &formatted_source_desc[total_char_processed]);
```

```
  strncat(source_details,curloc,char_remaining);
  for (i = strlen(source_details) ; i < 80; i++)
    source_details[i] = ' ';
  source_details[i+1] = '\0';
  strncat(source_details,"\n\n");
  if (WriteSourceDetails () != SUCCESS)
    return FAILURE;
  return SUCCESS;
}
/*********************************************************************
* Name        : WriteSourceDetails
*
* Description : Writes the source details to specified data file.
*
* Returns     : SUCCESS or FAILURE
*********************************************************************/
short WriteSourceDetails(void)
{
  FILE *dfp;                          /* pointer to source data file */
  if ((dfp = fopen(source_name_file,"a")) == NULL)
  {
    printf("execsu:ERROR: Failed to open file %s\n",source_name_file);
    return FAILURE;
  }
  fputs(source_details,dfp);
  fclose(dfp);
  return SUCCESS;
}
/*********************************************************************
* Name        : CheckWordSplit
*
* Description : Checks the source description record from the
*               location provided for number of characters that
*               will fit within a space reserved for 26 characters
*               in source description file without a word split
*               occurring at the end.
*
* Returns     : Number of characters to avoid a word split.
*********************************************************************/
short CheckWordSplit(char *line)
{
  short char_processed = 0 ;   /* characters in word                */
  short total_length =   0 ;   /* length of description read so far */
  char  word[2000];            /* word delimited by white space     */
  if (strlen(line) < 27)
    return strlen(line);
  while (TRUE)
  {
    /*
     * read the next word
```

```
     */
    sscanf(line,"%s",word);
    char_processed = strlen(word) +  1;
    if (char_processed > 26)
    {
      /*
       * found long word; return 25 and set global use_hypen to '1'
       */
      use_hypen = 1;
      return (25);
    }
    total_length= total_length + char_processed;
    line=line + char_processed ;
    if (total_length < 27)
      continue;
    else
    return (total_length - char_processed);
  }
}
/*********************************************************************
* Name        : StripLeadingSpaces
*
* Description : Strips leading spaces from a string
*
* Returns     : Pointer to string
*
* Notes       : 1 The variable total_char_processed is declared in
*                 ProcesssSourceDetails () and is treated as global
*
*               2 The variable total_char_processed is incremented
*                 for each leading space.
*********************************************************************/
char * StripLeadingSpaces (char *string)
{
  short i;
  short leading_space ;
  for (i= 0; i <strlen(string); i++)
  {
    if (*(string + i) != ' ')
    return (string + i) ;
    total_char_processed = total_char_processed + 1;
  }
}
/*********************************************************************
* Name        : WriteLog
*
* Description : The function writes a log message as follows:
*               o Action type WL writes a log message at the end
*                 of the log file.
*               o Action type UL writes a log message just below
```

```
*                 the previous one for the same source.
*
* Returns     : SUCCESS or FAILURE
*
* Notes       : 1 WL logs a message when source is checked out.
*
*               2 UL logs a message when source is checked in.
*
*               3 The log file is updated by reading the log file,
*                 writing it to a temporary file, and then copying
*                 the temporary file back to the log file.
*****************************************************************/
short   WriteLog (void)
{
  char   log_file[100];
  char   log_message[100];
  char   line[100];              /* line read from log file         */
  char   dummy[50];              /* word read from log file         */
  char   comment[30];            /* first word of the comment       */
  char   module[50];             /* source name read from the line  */
  char   p_module[20];           /* module as argument              */
  char   p_userid[20];           /* user id as argument             */
  short  log_updated = FALSE;
  short  i;
  char   temp_file [50];         /* temporary file for writing log  */
  char   system_command[100];
  short  rc;                     /* return code from system command */
  FILE   *lfp;                   /* pointer to log file             */
  FILE   *tfp;                   /* pointer to temporary file       */
  /*
   * prepare log file name
   */
  strcpy(log_file,dir_name);
  strcat(log_file,"/");
  strcat(log_file,file_name);
  /*
   * prepare temporary file name
   */
  strcpy(temp_file,"/tmp/");
  strcat(temp_file,"execsu_tmp.dat");
  /*
   * prepare the log message
   */
  strcpy(log_message,chk_in_out_date);
  strcat(log_message," ");
  strcat(log_message,chk_in_out_time);
  strcat(log_message,"    ");
  /*
   * turn user id into 15-character string
   */
```

```
strcpy(p_userid,chk_in_out_uid);
for (i=strlen(chk_in_out_uid);i < 15; i++)
  chk_in_out_uid[i] = ' ';
chk_in_out_uid[i]= '\0';
strcat(log_message,chk_in_out_uid);
/*
 * turn module name into fixed length of 22
 */
strcpy(p_module,chk_in_out_module);
for (i=strlen(chk_in_out_module);i<22; i++)
  chk_in_out_module[i] = ' ';
chk_in_out_module[i]= '\0';
strcat(log_message,chk_in_out_module);
strcat(log_message,chk_in_out_comment);
/*
 * open the log file in read  mode
 */
if ((lfp = fopen(log_file,"a")) == NULL)
{
  printf("execsu:ERROR: Failed to open file %s\n",log_file);
  return FAILURE;
}
if (strcmp(action_type,"WL") == 0)
{
  /*
   * write a log message at the end of file for the check-out
   */
  while (TRUE)
  {
    memset(line,'\0',100);
    fgets(line,100,lfp);
    if (strlen(line) == 0)
    {
      /*
       * end of file reached; write the new log message
       */
      strcat(log_message,"\n\n\n");
      fputs(log_message,lfp);
      fflush(lfp);
      fclose(lfp);
      return SUCCESS;
    }
  }
}
if (strcmp(action_type,"UL") == 0)
{
  /*
   * write a log message for the check-in below the corresponding
   * one for the check-out; open both log and temporary files
   */
```

```
if ((lfp = fopen(log_file,"r")) == NULL)
{
  printf("execsu:ERROR: Failed to open file %s\n",log_file);
  return FAILURE;
}
if ((tfp = fopen(temp_file,"w")) == NULL)
{
  printf("execsu:ERROR: Failed to open file %s\n",temp_file);
  return FAILURE;
}
/*
 * read and copy the header (5 lines)
 */
for (i=1; i<6; i++)
{
  memset(line,'\0',100);
  fgets(line,100,lfp);
  fputs(line,tfp);
}
while (TRUE)
{
  memset(line,'\0',100);
  memset(module,'\0',50);
  fgets(line,100,lfp);
  if (strlen(line) == 0)
    break;
  else
    fputs(line,tfp);
  /*
   * scan line for the checked-in module name
   */
  sscanf(line,"%s%s%s%s%s",dummy,dummy,dummy,module,dummy);
  if (strcmp(p_module,module) == 0)
  {
    /*
     * found CHECK OUT line, under which the new log message
     * is to be written; first we check whether the line
     * contains any other comments.
     */
    memset(line,'\0',100);
    memset(comment,'\0',30);
    fgets(line,100,lfp);
    if (strlen(line) == 0)
      break;
    sscanf(line,"%s%s%s%s%s",dummy,dummy,dummy,module,comment);
    if ((strcmp(comment,"UNMODIFIED") == 0) ||
        (strcmp(comment,"LOCKED") == 0))
    {
      /*
       * write the log line
```

```
           */
          fputs(line,tfp);
        }
        else
        {
          /*
           * found CHECK OUT line, under which the new log message
           * for check-in is written; first we check whether the line
           * contains any other comments.
           */
          strcat(log_message,"\n\n");
          fputs(log_message,tfp);
          fflush(tfp);
          log_updated = TRUE;
        }
      }
    }
    fclose(lfp);
    fclose(tfp);
    if (log_updated == TRUE)
    {
      /*
       * move the temporary file with updated log to log file
       */
      strcpy(system_command,"mv ");
      strcat(system_command,temp_file);
      strcat(system_command,"  ");
      strcat(system_command,log_file);
      rc = system(system_command);
      if (rc != SUCCESS)
      {
        printf("execsu:ERROR: Failed to copy temporary file %s\n",
               temp_file);
        return FAILURE;
      }
    }
    else
    {
      printf("execsu:ERROR: Failed to update log file %s\n",log_file);
      return FAILURE;
    }
  }
  return SUCCESS;
}
```

Arif Zamam
DBA/Administrator
High-Tech Software

Getting an entry from a configuration file

In a Windows system, it's fairly straightforward to extract an entry from a section of a configuration file. While this is very useful, the same is not so easy in an AIX environment. Hence, one misty morning, we decided to write a utility, called **get_entry**, that brings the same functionality to AIX. This allows you to make several configuration files and extract information from them.

EXAMPLE

Consider the following configuration file, called *system.ini*, which contains the following lines:

```
[tcpip]
host_address=140.88.76.5
host_name=AIX_system1
dns_server=140.20.6.2
ip_routing_active=no
use_proxy=no

[test_section]
user         = me and myself
script       =
rubbish      = yes
use_proxy    = always
```

The file comprises *entries* (lines that contain an equals sign, '='), and entries that are bounded by lines containing square brackets ('[' and ']') comprise *sections*. Each section begins with a word enclosed in square brackets and ends with a blank line. In this example we have two sections (the first has five entries and the second has four).

The following commands result in the following actions:

```
get_entry tcpip use_proxy system.ini
```

returns the word 'no' and the return code zero.

```
get_entry test_section user system.ini
```

returns the string 'me and myself' (without the single quotes) and the error code zero.

```
get_entry test_section not_to_find system.ini
```

returns no characters and the error code ‘1’ (‘entry not found’).

```
get_entry test_2 use_proxy system.ini
```

returns no characters and the error code ‘2’ (‘section not found’).

```
get_entry test_3 system.ini
```

returns the error code ‘3’ (‘missing parameter’) and the following lines:

```
Missing parameter(s):
  - section     : the section that contains the entry
  - entry       : the name of the entry to show
  - config file : the configuration file to be used
```

Example:

```
get_entry tcpip host_address /home/data/system.ini
```

Note the use of the continuation character, ‘➤’, in the code below to indicate that one line of code maps to more than one line of print.

GET_ENTRY

```
# Name script   : /home/oper/get_entry
# Last change   : 09-06-1997 pos9900  creation
# Description   : Get an entry from a section of a config file
# example       : get_entry subset1 script ini_file
#-----------------------------------------------------------------

# Check that we have all the required parameters
if [ $# -ne 3  -o "$1" = "?" ]
then if [ "$1" != "?" ]
     then echo "\nMissing parameter(s):"
     else echo "\n Help get_entry \n "
            echo "Required parameters:"
          fi
     echo "  - section     : the section that contains the entry"
     echo "  - entry       : the name of the entry to show"
     echo "  - config file : the configuration file to be used \n"
     echo "Example         : get_entry tcpip host_address
     ➤  /home/data/system.ini \n"
     exit 3
     fi

# Initialize : entry not found, but parameters are OK
entry_found=0
```

```
while read var1
      do
      # Check whether we've found the required subset
      if [ "$var1" = "[$1]" ]
      then # OK subset found
           entry_found=1
           fi

      # Empty line: another subset is coming
      if [ $entry_found -eq 1 -a  "$var1" = ""  ]
      then # Another subset, but entry still not found
           exit 1
           fi

      # Just looking for an entry in the subset
      if [ $entry_found -eq 1 ]
      then var2=`echo $var1 | awk -F= '{print $1}'`
           if [ "$var2" = "$2" -o "$var2" = "$2 " ]
           then # Got it, echo it and leave
                if [ "$var2" = "$2 " ]
                then # Entries formatted as 'entry    =  something'
                     echo $var1 | awk -F= '{print $2}' | cut -c2-
                else # Entries formatted as 'entry=something_else'
                     echo $var1 | awk -F= '{print $2}'
                     fi
                 exit 0
                fi
           fi
      done <$3

# Still searching for the subset, but end of file reached, so exit 2
exit 2

# end script
```

Teun Post
Unix specialist
Schuitema NV (The Netherlands)

Removing files by timestamp

It would be nice to have a command that removes files from a given directory based on a specified date and time. **refbot.sh** is a shell script that accomplishes this task. The script removes from a specified directory all the files that were modified before the date specified as the argument to the script. The script can be run in interactive mode, where the script seeks the user's approval prior to removing files.

OVERVIEW

Input:

1 **D=<directory name>**

This input is mandatory.

2 **d=<ddmmyyyyhhmiss>**

Date and time. This input is mandatory.

3 **i=<y\n>**

Interactive flag. This input is optional (the default is 'Y').

4 **l=<logfile name>**

Log file name. This input is optional (the default is */tmp/refbot.log*).

USAGE

Note the use of the continuation character ('➤') below to indicate that one line of code maps to more than one line of print.

```
refbot.sh  D=/home/jones_e/work  d=13081999120000  i=y
➤          l=/admin/refbot.log
```

PROCESSING

Below is an overview of the structure of the program, which may prove useful to those wishing to customize the utility to their own needs.

```
Switch to specified directory
Prepare a list of all the files
FOR each file
   Establish whether file qualifies for removal
   IF the file is to be removed
      IF the file owner is the current user or the current
user is root
         IF interactive mode is on
            IF user approves to removal
               IF file is directory
                  remove the directory with contents
               ELSE
                  Remove the file
                  Write a log
               END IF
            END IF
         ELSE
            Remove the file
            Write a log
         END IF
      END IF
   END IF
```

REFBOT.SH

```
#! /usr/bin/ksh
###############################################################################
# Name     : refbot.sh (remove file based on time)
#
# Overview : The script removes file(s) from a given directory
#            based on date and time.
#
# Input    : 1 Directory name (no default)
#            2 Date and Time (no default)
#            3 Interactive flag (default to Y)
#            4 log file (default to /tmp/refbot.log)
#
# Notes    1 The script removes the file(s) which were modified or
#            accessed prior to input date and time.
#
#          2 The syntax for the command is as follows:
#            refbot.sh D=<directory name> t=<ddmmyyyyhhmiss>
#                      i=<y|n> l=<file name>
#
#            eg refbot.sh D=/tmp t=01081999120022 i=n
#                         l=/home/admin/log/refbot.log
#
#          3 The script contains the following functions:
#            o main
```

```
#             o InitializeVariables
#             o ParseCommandLine
#             o ValidateArguments
#             o LeapYear
#             o RemoveFiles
#             o FileToBeRemoved
#             o FileQualifyForRemoval
#             o ProcessExit
#             o HandleInterrupt
#             o DisplayMessage
#
#          4 The user must acknowledge all displayed messages.
#
#          5 Files or directories not owned by ordinary user are not
#            removed. However, all the files and/or directories that
#            qualify for removal will be removed if the user is
#            root.
#
#          6 If a directory qualifies for removal, the utility uses
#            the rm -r command to remove the directory and its
#            contents.
#
# Date       Author   Description
# ---------------------------------------------------------------
# 09/08/99  A Zaman  Initial Build
#
# 26/08/99  A Zaman  Improved handling of directory removal
#
################################################################################

################################################################################
# Name     : InitializeVariables
#
# Overview : Initializes all required variables.
################################################################################
InitializeVariables ()
{
# extract user id
USERID=`id | tr "()" "::" | cut -d':' -f2`

# define date and time
DATETIME=`date "+%d/%m/%Y at %H:%M:%S"`

# define temporary files
FILE_LIST=/tmp/refbot_$$_1.dat
QUALIFIED_FILE_LIST=/tmp/refbot_$$_2.dat
TEMP_FILE_1=/tmp/refbot_$$_1.tmp
TEMP_FILE_2=/tmp/refbot_$$_2.tmp

# Log file
```

```
DEFAULT_LOG_FILE="/tmp/refbot.log"

# Return codes
TRUE=0
FALSE=1
SEC=0
FEC=1
# Define escape sequences
ESC="\0033["
RVON=_[7m            # Reverse video on
RVOFF=_[27m          # Reverse video off
BOLDON=_[1m          # Bold on

BOLDOFF=_[22m        # Bold off
BON=_[5m             # Blinking on
BOFF=_[25m           # Blinking off

# Input parameter variables
DIR_NAME=            # Directory name
DATE=                # Date and time
INTERACTIVE=         # Interactive flag
LOG_FILE=            # Log file

SLEEP_DURATION=3     # Number of seconds for sleep
ERROR="${RVON}${BON}refbot.sh:ERROR:${BOFF}"
INFO="${RVON}refbot.sh:INFO: "

# Messages
WORKING="Working...${RVOFF}"
INTERRUPT="Program interrupted! Quitting...${RVOFF}"
ROOT_USER="Script must be executed from root account${RVOFF}"
WORKING="Working...${RVOFF}"
USAGE="Usage\:refbot.sh D=\<directory name\> d=\<date\>
➤  i=\<y\|n\> l=\<file\>${RVOFF}"
DIR_REQ="Please provide a directory using option D${RVOFF}"
DATE_REQ="Please provide a date using option d${RVOFF}"
DUP_ARG="Duplicate argument${RVOFF}"
INVALID_ARGC="Wrong number of arguments${RVOFF}"
INVALID_ARG_TYPE="\${ARG_TYPE}, is an invalid argument${RVOFF}"
INVALID_OPTION="\${OPTION}, is an invalid option${RVOFF}"
INVALID_ENTRY="Invalid entry${RVOFF}"
INVALID_DATE="Invalid date${RVOFF}"
INVALID_DIR="Invalid directory, \${DIR_NAME}${RVOFF}"
INVALID_TIME="Invalid time${RVOFF}"
INVALID_MONTH_LEN="Month must be in two-digit format${RVOFF}"
INVALID_DAY_LEN="Day must be in two-digit format${RVOFF}"
INVALID_YEAR_LEN="Year must be in four-digit format${RVOFF}"
INVALID_HOUR_LEN="Hour must be in two-digit format${RVOFF}"
INVALID_MIN_LEN="Minute must be in two-digit format${RVOFF}"
INVALID_SEC_LEN="Second must be in two-digit format${RVOFF}"
```

```
INVALID_DAY="Invalid day${RVOFF}"
INVALID_MONTH="Invalid month${RVOFF}"
INVALID_YEAR="Invalid year${RVOFF}"
INVALID_HOUR="Invalid hour${RVOFF}"
INVALID_MINUTES="Invalid minutes${RVOFF}"
INVALID_SECONDS="Invalid second${RVOFF}"
DATE_NOT_NUMERIC="Date, \${DATE} must be numeric${RVOFF}"
TIME_NOT_NUMERIC="Time, \${TIME} must be numeric${RVOFF}"
INVALID_FLAG="\${INTERACTIVE}, is an invalid value for option
➤  i${RVOFF}"
LOG_NOT_INITIALIZED="Failed to initialize log file,
➤  \${LOG_FILE}${RVOFF}"
LOG_NOT_WRITABLE="Log file, \${LOG_FILE} is not writable by
➤  user${RVOFF}"
OS_ERROR="\${SYSERROR}${RVOFF}"
DIR_NOT_ACCESSABLE="Directory, \${DIR} is not accessible${RVOFF}"
DIR_EMPTY="Directory, \${DIR} is empty${RVOFF}"
NO_FILE_REMOVED="No file qualifies for removal in directory,
➤  \${DIR} ${RVOFF}"
REQ_USER="Must execute the script from root account${RVOFF}"

# Signals
SIGNEXIT=0 ; export SIGNEXIT # normal exit
SIGHUP=1 ; export SIGHUP # session disconnected
SIGINT=2 ; export SIGINT # ctrl-c
SIGTERM=15 ; export SIGTERM # kill command
SIGTSTP=18 ; export SIGTSTP # ctrl-z
}

##############################################################################
# Name     : HandleInterrupt
#
# Overview : Call ProcessExit.
##############################################################################
HandleInterrupt ()
{
DisplayMessage I "${INTERRUPT}"

ProcessExit $FEC
}

##############################################################################
# Name     : MoveCursor
#
# Input    : Y and X coordinates
#
# Overview : Moves the cursor to the required location (Y, X).
#
# Notes    1 The function must run in ksh for print to work. Print
#            is used to move the cursor as echo doesn't work.
```

```
##############################################################################
MoveCursor ( )
{
trap "HandleInterrupt" $SIGINT $SIGTERM $SIGHUP $SIGTSTP

YCOR=$1
XCOR=$2

echo "${ESC}${YCOR};${XCOR}H"
}

##############################################################################
# Name     : DisplayMessage
#
# Overview : Display message
#
# Input    : 1 Message type (E = Error, I = Information)
#            2 Error Code, defined in DefineMessages ().
#            3 Message to acknowledge flag
#
# Notes    1 User must acknowledge all messages except 'WORKING'.
##############################################################################
DisplayMessage ( )
{
trap "HandleInterrupt" $SIGINT $SIGTERM $SIGHUP $SIGTSTP

MESSAGE_TYPE=$1
MESSAGE_TEXT=`eval echo $2`
ACK_MESSAGE="$3"
if [ "${ACK_MESSAGE}" = "" ]
then
      ACK_MESSAGE="Y"
fi
clear
MoveCursor 24 1
if [ "${MESSAGE_TYPE}" = "E" ]
then
   echo "`eval echo ${ERROR}`${MESSAGE_TEXT}\c"
else
   echo "`eval echo ${INFO}`${MESSAGE_TEXT}\c"
fi

# Let user acknowledge the message
if [ "${ACK_MESSAGE}" = "Y" ]
then
   read DUMMY
fi
return ${TRUE}
}
```

```
#############################################################################
# Name     : ProcessExit
#
# Overview : Processes menu options.
#############################################################################
ProcessExit ()
{
EXIT_CODE="$1"

rm -f $FILE_LIST
rm -f $QUALIFIED_FILE_LIST
rm -f $TEMP_FILE_1
rm -f $TEMP_FILE_2

clear

exit ${EXIT_CODE}
}

#############################################################################
# Name     : LeapYear
#
# Overview : Establishes whether a given year is a leap year.
#
# Input    : Year
#
# Returns  : $TRUE for leap year
#            $FALSE otherwise
#############################################################################
LeapYear ()
{
trap "HandleInterrupt" $SIGINT $SIGTERM $SIGHUP $SIGTSTP

# Assign the parameter
P_YEAR="$1"

# Divide $P_YEAR by 4 to establish leap year
RESULT=`bc <<!
scale=2
$YEAR/4
!`
if [ "`echo $RESULT | cut -d'.' -f2`" = "00" ]
then
   # year is leap year
   return $TRUE
else
   # year is not a leap year
   return $FALSE
fi
}
```

```
#############################################################################
# Name     : ValidateArguments
#
# Overview : Validate arguments values.
#
# Returns  : $TRUE or $FALSE
#
# Notes    1 The function validates the following arguments' values:
#            o directory (mandatory)
#            o date (mandatory)
#            o interactive flag (optional)
#            o log file (optional)
#############################################################################
ValidateArguments ()
{
trap "HandleInterrupt" $SIGINT $SIGTERM $SIGHUP $SIGTSTP

# Validate directory
if [ -z "${DIR_NAME}" ]
then
   DisplayMessage E "${DIR_REQ}"
   return $FALSE
fi
if [ ! -d $DIR_NAME ]
then
   DisplayMessage E "${INVALID_DIR}"
   return $FALSE
fi

# Validate date (eg DDMMYYYYHHMISS)
if [ -z "${DATE}" ]
then
   DisplayMessage E "${DATE_REQ}"
   return $FALSE
fi

# Validate numeric
if ( [ `expr $DATE + 0` -eq $DATE ] ) > /dev/null 2>&1
then
      : # ok
else
      DisplayMessage E "${DATE_NOT_NUMERIC}"
      return $FALSE
fi

# Validate overall length
LEN=`expr "$DATE" : '.*'`
if ! ( [ $LEN -eq 14 ] )
then
```

```
   DisplayMessage E "${INVALID_DATE}"
   return $FALSE
fi

# Verify the date format is <ddmmyyyy>
DAY=`echo "$DATE" | cut -c1-2`
MON=`echo "$DATE" | cut -c3-4`
YEAR=`echo "$DATE" | cut -c5-8`

# Validate day
LEN=`expr "$DAY" : '.*'`
if ! ( [ $LEN -eq 2 ] )
then
   DisplayMessage E "${INVALID_DAY_LEN}"
   return $FALSE
fi
if ! ( [ $DAY -gt 0 -a $DAY -lt 32 ] )
then
      DisplayMessage E "${INVALID_DAY}"
      return $FALSE
fi

# Validate month
LEN=`expr "$MON" : '.*'`
if ! ( [ $LEN -eq 2 ] )
then
   DisplayMessage E "${INVALID_MONTH_LEN}"
   return $FALSE
fi
if ! ( [ $MON -gt 0 -a $MON -lt 13 ] )
then
      DisplayMessage E "${INVALID_MONTH}"
      return $FALSE
fi

# Validate year
LEN=`expr "$YEAR" : '.*'`
if ! ( [ $LEN -eq 4 ] )
then
   DisplayMessage E "${INVALID_YEAR_LEN}"
   return $FALSE
fi
if ! ( [ $YEAR -gt 0 ] )
then
   DisplayMessage E "${INVALID_YEAR}"
   return $FALSE
fi

# Validate month
LEN=`expr "$MON" : '.*'`
```

```
if ! ( [ $LEN -eq 2 ] )
then
   DisplayMessage E "${INVALID_MONTH_LEN}"
   return $FALSE
fi
if ! ( [ $MON -gt 0 -a $MON -lt 13 ] )
then
   DisplayMessage E "${INVALID_MONTH}"
   return $FALSE
fi

# Validate year
LEN=`expr "$YEAR" : '.*'`
if ! ( [ $LEN -eq 4 ] )
then
   DisplayMessage E "${INVALID_YEAR_LEN}"
   return $FALSE
fi
if ! ( [ $YEAR -gt 0 ] )
then
   DisplayMessage E "${INVALID_YEAR}"
   return $FALSE
fi

# Validate day and month (other than February)
if [ $MON -eq 01 -o $MON -eq 03 -o $MON -eq 05 -o \
     $MON -eq 07 -o $MON -eq 08 -o $MON -eq 10 -o \
     $MON -eq 12 ]
then
   if ! ( [ $DAY -gt 0 -a $DAY -lt 32 ] )
   then
         DisplayMessage E "${INVALID_DAY}"
         return $FALSE
   fi

elif [ $MON -eq 04 -o $MON -eq 06 -o $MON -eq 09 -o \
       $MON -eq 11 ]
then
   if ! ( [ $DAY -gt 0 -a $DAY -lt 31 ] )
   then
         DisplayMessage E "${INVALID_DAY}"
         return $FALSE
   fi
fi

# Validate month of February
if LeapYear "${YEAR}"
then
   if [ $MON -eq 02 ]
   then
```

```
      if ! ( [ $DAY -gt 0 -a $DAY -lt 30 ] )
      then
         DisplayMessage E "${INVALID_DAY}"
         return $FALSE
      fi
   fi
else  #
      # Not a leap year
   if [ $MON -eq 02 ]
   then
      if ! ( [ $DAY -gt 0 -a $DAY -lt 29 ] )
      then
         DisplayMessage E "${INVALID_DAY}"
         return $FALSE
      fi
   fi
fi

# Validate time (eg HHMMSS)

# Verify the date format is <ddmmyyyyhhmiss>
HOUR=`echo "$DATE" | cut -c9-10`
MIN =`echo "$DATE" | cut -c11-12`
SEC =`echo "$DATE" | cut -c13-14`

# Valixdate hours
LEN=`expr "$HOUR" : '.*'`
if ! ( [ $LEN -eq 2 ] )
then
   DisplayMessage E "${INVALID_HOUR_LEN}"
   return $FALSE
fi

if ! ( [ $HOUR -eq 0 -o $HOUR -lt 24 ] )
then
   DisplayMessage E "${INVALID_HOUR}"
   return $FALSE
fi

# Validate minutes
LEN=`expr "$MIN" : '.*'`
if ! ( [ $LEN -eq 2 ] )
then
   DisplayMessage E "${INVALID_MIN_LEN}"
   return $FALSE
```

```
fi
if ! ( [ $MIN -eq 0 -o $MIN -lt 60 ] )
then
   DisplayMessage E "${INVALID_MINUTES}"
   return $FALSE
fi

# Validate seconds
LEN=`expr "$SEC" : '.*'`
if ! ( [ $LEN -eq 2 ] )
then
   DisplayMessage E "${INVALID_SEC_LEN}"
   return $FALSE
fi
if ! ( [ $SEC -eq 0 -o $SEC -lt 60 ] )
then
   DisplayMessage E "${INVALID_SECONDS}"
   return $FALSE
fi

# Validate interactive flag
case ${INTERACTIVE} in
   y|Y|n|N ) : ;;
      "" ) INTERACTIVE="Y";;
       * ) DisplayMessage E "${INVALID_FLAG}" ;
           return $FALSE ;;
esac

# Validate log file
if [ "${LOG_FILE}" = "" ]
then
   LOG_FILE="${DEFAULT_LOG_FILE}"
fi

return $TRUE
}
```

This article concludes in next month's issue of *AIX Update*.

Arif Zaman
DBA/Administrator
High-Tech Software Ltd (UK)

Softworks has announced SST-Resource Availability Version 3.4, which monitors and reports on storage resources. Among the improvements in the new version are better reporting of storage utilization and wider platform coverage for Oracle and ADSM. Also new is the measurement of storage consumption at the user or application level.

In addition to AIX, the software supports logical and physical storage systems used by OS/390, MVS, a number of variants of Unix, and NT. Out now, prices start at US$57,000.

For further information contact:
Softworks Computer Concepts, 5845 Richmond Highway, Suite 200, Alexandria, VA 22303, USA
Tel: +1 703 317 2424
Fax: +1 703 317 3229
Web: http://www.softworkscc.com

Softworks International Limited, Clayton House, 3-7 Vaughan Road, Harpenden, Hertfordshire, AL5 4EF, England
Tel: +44 1582 464800
Fax:+44 1582 767941

* * *

Candle has launched Roma E-business Platform 2000, part of its CandleNet family of e-business lifecycle products that also includes Service Provider Platform and E-Business Assurance Network.

Roma E-business Platform is an application 'integration backplane' into which applications can be 'plugged'. This helps reduce the new development overhead associated with fielding new functionality. The platform supports Java, C, C++, COBOL, and COM/CORBA. Applications can be both XML-compliant and LDAP-enabled. Besides AIX, operating system support includes OS/390, AS/400, Solaris, and NT.

The product is expected in the first quarter of 2000 and details on pricing are available on request from the vendor.

For further information contact:
Candle Corp, 2425 Olympic Blvd, Santa Monica, CA 90404, USA
Tel: +1 310 829 5800
Fax: +1 310 582 4287
Web: http://www.candle.com

Candle Ltd, 1 Archipelago, Lyon Way, Frimley, Camberley, Surrey GU16 5ER, UK
Tel: +44 1276 4147000
Fax: +44 1276 414777

* * *

IBM has announced DCE Version 3.1 for AIX and Solaris. Features include a common code base between the two OS platforms, Kerberos V5 interoperability, DCE audit information enhancements, password strength enhancements, public key certificate login, and Global Directory Agent (GDA) using the Lightweight Directory Access Protocol (LDAP).

Out now, it costs US$4,000.

For further information contact your local IBM representative.