76

AIX update

February 2002

In this issue

3 Sifting through the new pSeries hardware options – SP2 or not to SP2?

7 An introduction to emacs – part 2

12 Processes explained

18 Introduction to the shell

21 AIX 5L routing enhancements

30 Shell-scripting tricks

48 AIX news

© Xephon plc 2002

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.

© Xephon plc 2002. All rights reserved. None of the text in this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior permission of the copyright owner. Subscribers are free to copy any code reproduced in this publication for use in their own installations, but may not sell such code or incorporate it in any commercial product. No part of this publication may be used for any form of advertising, sales promotion, or publicity without the written permission of the publisher. Copying permits are available from Xephon in the form of pressure-sensitive labels, for application to individual copies. A pack of 240 labels costs $36 (£24), giving a cost per copy of 15 cents (10 pence). To order, contact Xephon at any of the addresses above.

Printed in England.

Sifting through the new pSeries hardware options – SP2 or not to SP2?

The new hardware announcements from IBM can make it difficult to decide a direction for RS/6000 SP2 systems growth and additions. I began sifting through the announcements and questioning my third-party vendor(s) for information on the new hardware, and ended up with a fair amount of information to share.

My slant on gathering this information is not only which pSeries machine to obtain or evaluate, but whether or not to continue to add nodes to my SP2 environment, or to add externally PSSP clustered boxes, or to abandon PSSP altogether for new projects. This decision is further complicated by the fact that some of the new pSeries rack-mountable machines cannot participate in PSSP clustering. This last 'catch' is somewhat disappointing to me because I have grown to appreciate many of the PSSP benefits, even though admittedly there is a fairly large learning curve for new administrators, depending on the complexity of your environment.

REVIEW OF PSSP BENEFITS

PSSP has many value-added benefits over and above standard AIX, including central hardware control and monitoring, Virtual Shared Disks and recoverable VSDs, the SP switch, the system data repository, centralized user and printer management, parallel programming options, GPFS for filesystem sharing, and so on. The argument to continue to use PSSP is a simple one.

For the current pSeries server specifications see Figure 1.

The focus of this chart is new rack-mountable pSeries machines. There are a number of other 'legacy' models that are still available at this point. As you can see, the SP2 node options, ie nodes to go in the frame, have become limited. They compare loosely to between a high-end 640 and mid-size 660, once everything is taken into account. If you need a server with requirements that fall well below or above this

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

Figure 1: Current pSeries server specifications

Server	CPU Speed(s)	Processors	Type	SP - Attach
pSeries 610	375/450 MHz	1 or 2	P3-II	NO
pSeries 640	375/450 MHz	1,2 or 4	P3-II	NO
pSeries 660 – 6H0	450/600 MHz	1,2, or 4	RS64 III/IV	YES
pSeries 660 – 6H1	450/600 MHz	1,2,4 or 6	RS64 III/IV	YES
pSeries 660 – 6M1	500/750 MHz	2,4,6 or 8	RS64 III/IV	YES
pSeries 680	450/600 MHz	4,6,12,18 or 24	RS64 III/IV	YES
pSeries 690	1.1/1.3 GHz	8,16,24,32	Power4	YES(2002)
SP node Thin	375 MHz	2 or 4	Power3-II	YES
SP node Wide	375 MHz	2 or 4	Power3-II	YES
SP node High	375 MHz	4,8,12 or 16	Power3-II	YES

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

range, an SP2 Frame node is no longer an option. Note: as far as specs go, there is a wealth of information for these servers and nodes at the IBM Web site, http://www-1.ibm.com/servers/eserver/pseries/hardware/factsfeatures.html.

MAKING A DECISION TO GO WITH AN 'EXTERNAL' NODE

Once you realize that one of the other pSeries models is more appropriate for your project than the available SP2 nodes, there are some things to consider:

- Switch – if you have a switch in your current SP2 environment, and you want to control your new external node with PSSP, then it will have to be switch-attached. This now becomes a question of available switch ports. The traditional 100MB SP switch has 16 ports. Any free ports can be used to switch-attach the external node. However, if your switch is fully used, this becomes a quandary: you have to either purchase another switch or re-consider using PSSP on the new node. For example, if you have a 1-Frame SP2 with a switch, and eight wide nodes in the frame, eight switch ports are available. If you have 2-Frames, a total of 12 wide nodes, for example, and one switch, then you have four switch ports available. In this latter example, if there were a total of 16 wide nodes and one switch, the purchase of a second switch would be required to make available more switch ports. The variations on this are many and need to be examined carefully.
- PSSP software licence fee – a PSSP licence must be obtained for the externally PSSP-controlled node. This cost is not a factor for 'normal' SP2 nodes. The cost of this licensing, and the switch adapter, can quickly wipe out any cost savings found in going with the external node.
- Alternative partition – if you did not want to invest in another switch, another approach would be to start another 'PSSP Partition' of just external PSSP nodes. This would require an additional control workstation (this could be a 'lightweight' box).

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

Depending on your needs for communication to some of the SP2 nodes you could then consider having a closed high-speed network (ATM or GB Ethernet for example) between them.

SUMMARY

While it appears that the line of nodes available for SP2 frames has come to the end of its evolution, there are plenty of alternatives to continue to exploit PSSP and its benefits. Most research on the future of PSSP indicates that it will be around in some form, probably 'natively' in AIX in the future, for a long time.

SPECIAL NOTES ON PSERIES 610 AND 690

The pSeries 610 and 690 machines, both recently announced, have a couple of unique qualities that do not necessarily pertain to PSSP/SP2, but are interesting to mention.

The 610, which is a small entry-level server that is available in a tower or rack mounted, can boot Linux natively, as opposed to the Linux affinity inherent in AIX 5L.

This is a first for RS/6000s.

The 690 is the first server to use the new POWER4 architecture, and the first RS/6000 that will support logical partitioning. LPARs will allow you to have multiple instances of AIX running in the same physical box. For example you could have an 'online LPAR' and a 'batch LPAR', both of which could have CPU and memory resources allocated to them accordingly. While the 690 is a large enterprise-class server that many of our budgets will never stretch to, rumour has it that LPARs will be supported on a number of new 'smaller' POWER4 servers due to be announced early in 2002.

David Miller
Database Architect
Baystate Health Systems (USA)

© Xephon 2002

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

An introduction to emacs – part 2

This month we conclude our introductory look at emacs.

Search and replace

Emacs has several powerful ways to find the bit of text you require. By far the favourite, however, is for simple searches that do not require regexp matching, and is the Incremental Search (I-search), bound to *C-s* (for looking forwards) and *C-r* (for looking backwards). Move to the start of a document, and press *C-s*. Type in 'Search'. As you type, emacs jumps to the next word in the document after the current location, where the word contains the letter(s) you've typed thus far. So after pressing *C-s S*, you will have moved to the first instance of 'S' in this file. As you proceed to enter more characters, the more refined the search becomes, so you start matching 'Sear', which might only match 'Search'. If you press *C-s* again during an on-going search, it jumps to the next occurrence of the current search-string, so you can bounce forwards through the document looking for 'Sear' if you want to. Likewise, *C-r* behaves identically apart from working backwards. Use a cursor or other motion key to stop I-searching.

Replacing text is generally best performed by using regular expressions. For example, if you were viewing this document in HTML form, you could replace all the <h3> tags with <h4> by doing the following:

```
M-x replace-regexp RET <h3> RET <h4> RET
```

and you'll find all occurrences, from the current point downwards, become replaced. This is similar to:

```
:%s/<h3>/<h4>/g
```

in vi, except that emacs' regular expressions are slightly more complicated and differently behaved.

This is one small difference between emacs and Xemacs; by default, Xemacs21 will recognize a highlighted region, restricting its search

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

and replace activities to this region, while in emacs20 and emacs21 you need to enable transient-mark-mode to have the same effect.

It is possible to perform a query search and replace, by using *M-%*. This is similar to ‘replace string’, except that at each found instance, emacs queries whether you want to replace it.

Keyboard macros

Emacs has the ability to store keyboard events as macros for later batch execution. To start recording a macro, press *C-x (*. Perform as many keyboard operations as you wish, moving text around, inserting things, and then close off the macro with *C-x)*. You can now repeat the same activities by pressing *C-x e*. To repeat it more than once, use a prefix argument, such as *C-u 10 C-x e*.

An example, in full: take an ordinary HTML document, move to the start of the last line, and type:

```
C-x ( UP C-e SPACE C-d C-a C-x )
```

The mini-buffer will now display ‘Keyboard macro defined’, and the buffer contents will have changed such that the last two lines have been joined together, and (importantly!) the cursor will be left at the start of the previous line. Now repeat this as many times as there are lines in the buffer, so that the whole document is one long line.

To restore the buffer to a semi-sanitized content, define another keyboard macro, thus:

```
C-x ( C-s < RIGHT LEFT ENTER RIGHT C-x )
```

This will I-search forwards for the next ‘<’ character, and immediately before it, insert a new line, (importantly!) leaving the cursor just after the ‘<’ character in question.

For an alternative method to achieve the above, you can also use replace-regexp to replace all instances of *C-j* by the empty string (use *C-q C-j* to obtain a literal newline in the mini-buffer; note that this will probably be displayed by changing the mini-buffer to a blank line while you type it, except in emacs21, which has a multi-line mini-buffer by default), and then perform a search and replace back again, to replace ‘<’ with *C-q C-j <*.

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

Either method has its benefits; it's normally hard work to debug a long regexp if it doesn't work the first time, but it can also be time-consuming to retype a whole macro if the actions it performs are very complex.

Configuration

Emacs stores its per-user configuration, a series of LISP s-expressions, in a file, ~/.emacs (note: not .emacsrc). For those not yet happy with emacs LISP, there is a handy customize interface in emacs21 and Xemacs, which you can enter by typing *M-x customize* – see Figure 1. In emacs20, you'd be best off learning LISP.

Within these, you can press TAB to jump to the next link, *Enter* to enter a section, *q* to quit the buffer (cancel configuration), and press the Set, Save, and Done buttons to assert a configuration. Now, preserve it in your config file, and leave the configuration buffer.

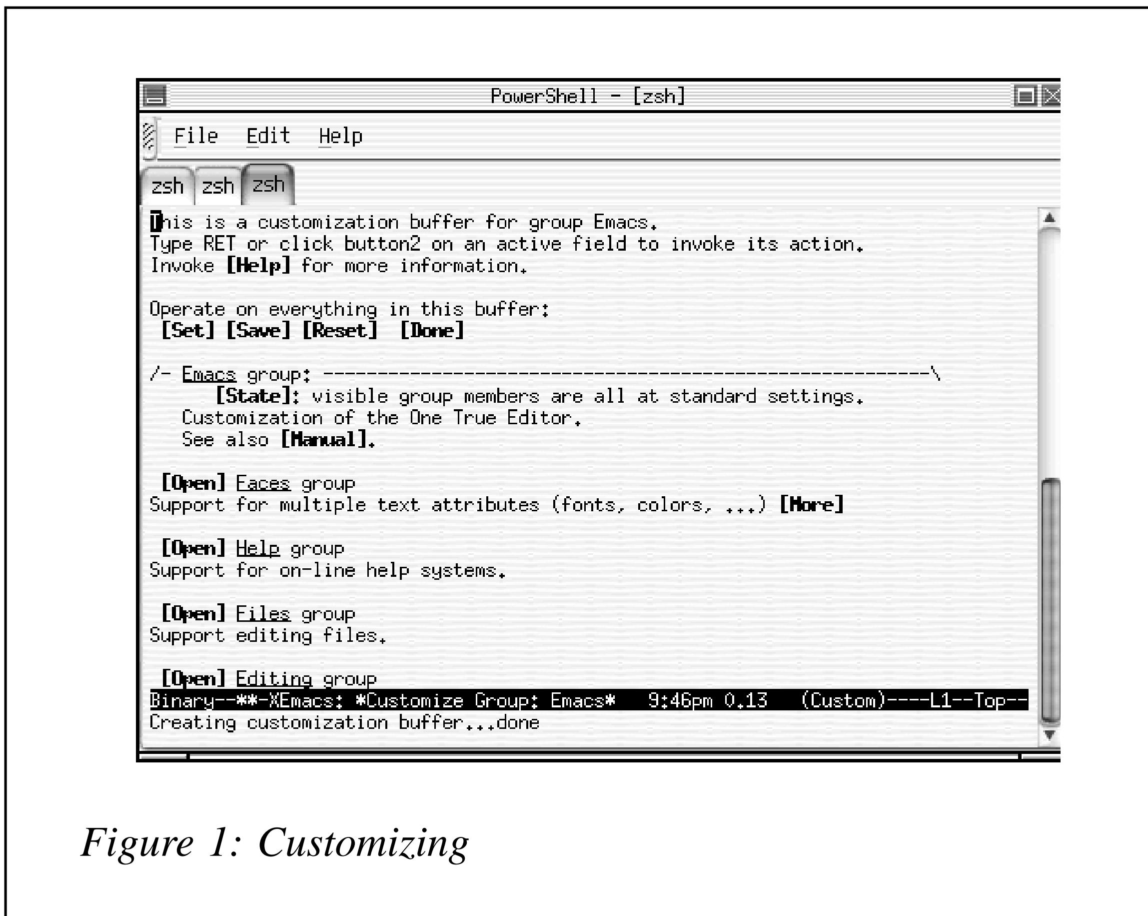

Figure 1: Customizing

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

Here is an example of what customize should write into your ~/.emacs file:

```
(custom-set-variables
  ;; custom-set-variables was added by Custom — don't edit or cut/paste
it!
  ;; Your init file should contain only one such instance.
 '(article-hide-headers t)
 '(backup-by-copying nil)
 '(bell-volume 0)
 '(browse-url-browser-function (quote browse-url-w3) t)
 '(current-language-environment "ASCII")
 '(debug-on-error nil)
 '(fast-lock-mode t t (fast-lock))
 '(filladapt-mode t t (filladapt))
 '(font-lock-mode t t (font-lock))
 '(frame-background-mode nil)
 '(global-font-lock-mode t nil (font-lock)))
```

Reader exercise

Use the following:

```
C-u M-! ls -1 | grep -v ' '
```

to insert a list of the files (excluding those with any spaces in the name) in the current directory into the buffer. Use the emacs help system to find out what *M-!* does, in particular what the purpose of the leading *C-u* was.

Now use replace-regexp to convert the start of each line (‘^’) into the string:

```
<a href="
```

ie the start of an HTML <a> tag.

Now devise a keyboard macro using *C-k* and *C-y* (amongst others) to complete each of the lines, so that each file becomes a link to itself, thus:

```
<a href="intro.dvi">intro.dvi</a>
<a href="intro.html">intro.html</a>
<a href="intro.latte">intro.latte</a>
<a href="intro.pdf">intro.pdf</a>
<a href="intro.ps">intro.ps</a>
<a href="intro.tex">intro.tex</a>
<a href="intro.txt">intro.txt</a>
<a href="xemacs_and_emacs_modes.png">xemacs_and_emacs_modes.png</a>
```


© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

Dired

If you open a directory as though it were a file, by typing something like *C-x C-f C-a ~ RET*, for example, you will see a new buffer, a sort of file manager of buffers. Within this, you can highlight files, press RET to open each file in its own buffer, or press *d* to mark them for deletion (then *X* to commit the deletions).

CLOSING TASTER – EMACS LISP

You will have seen references to ‘functions’ and LISP above.

Emacs is based entirely on its own native dialect of LISP (see www.LISP.org). This shows through in several ways, but most notably in the fact that you can find out what pressing any key (combination) does, and the answer always comes back in terms of a function. Right

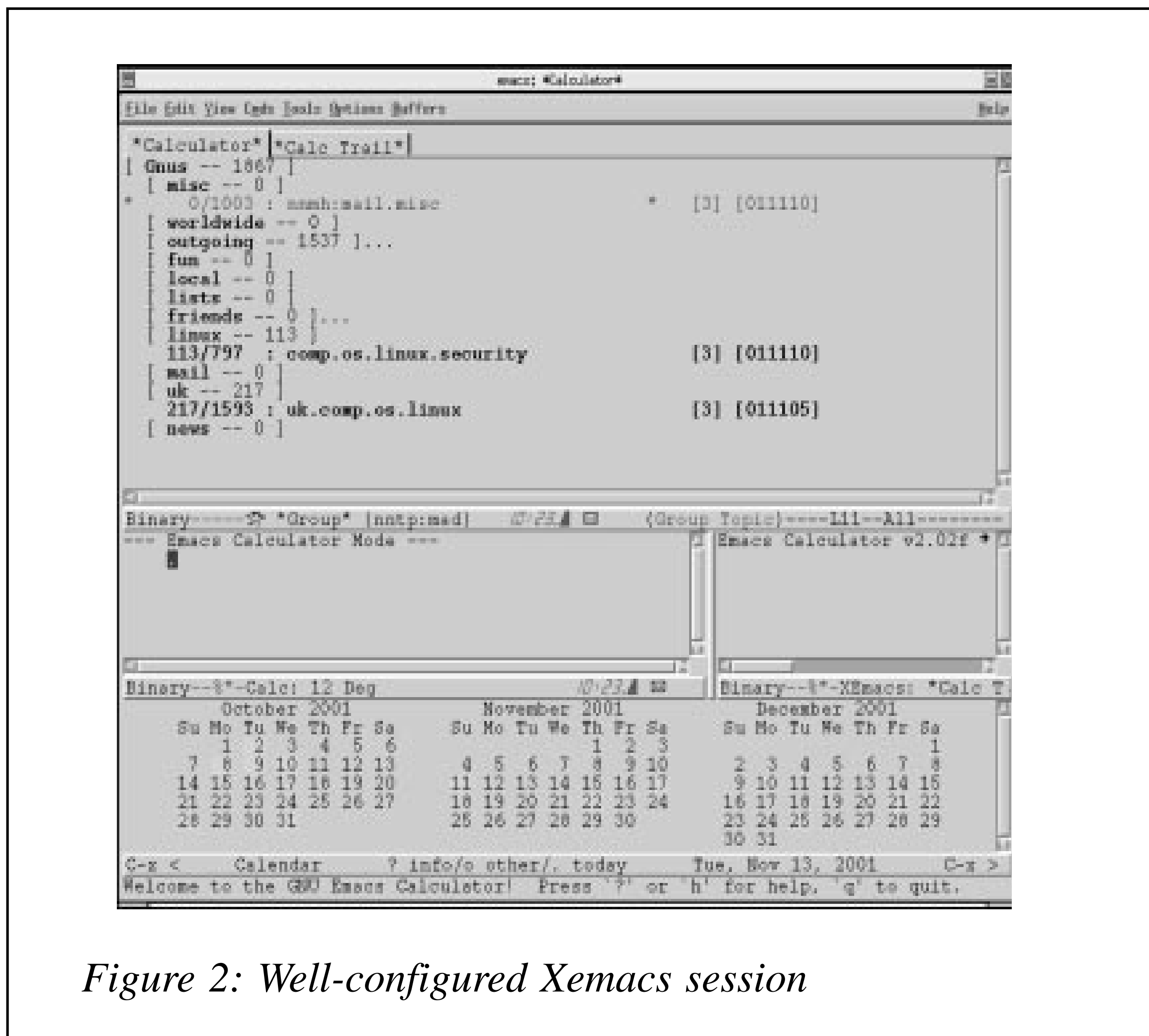

Figure 2: Well-configured Xemacs session

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

now, *C-h k e* tells me that ‘e’ is bound to the ‘self-insert-command function’, which inserts a character corresponding to the key pressed. Try it. Try doing *C-h k C-h k* as well.

For a picture of a reasonably well-configured Xemacs session showing Gnus, the calculator, and the calendar all at once, see Figure 2.

We have already seen references to *M-x*. This invokes a **command** in emacs LISP, which is a function defined to be **interactive**. As an example, consider this (from *C-h k M-q*):

```
M-q runs 'fill-paragraph-or-region'
[snip]
Documentation:
Fill the current region, if it's active; otherwise, fill the paragraph.
See 'fill-paragraph' and 'fill-region' for more information.
```

This means that, if you were to type *M-x fill-paragraph-or-region*, it would have the same effect as pressing *M-q*.

There is also *M-:*, which evaluates a given LISP expression, putting the results in the mini-buffer.

For example, try doing this:

```
M-: (+ 3 (- 2 9) (* 13 4))
```

You should see the result, 48, in the mini-buffer.

To run a quick one-liner that inserts the results of an expression into the current buffer, try:

```
M-: (insert (current-time-string))
```

Tim Haynes
Open Source and Free Software Consultant (UK) © Xephon 2002

Processes explained

In AIX (and other flavours of Unix), a process is the means by which a program gets executed. Any program that needs to be run on a machine managed by Unix must be run in a process, which is a collection of system resources.

 © 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

A running program in a process can be replaced by a new program.

PROCESS CREATION

Process creation is a system activity. When we try to run a program, a process is automatically created by the system, using the system call **fork ( )**. If there is not enough memory in the system this process creation can fail. If the creation of a new process will exceed the process creation limit allocated to the system user, process creation will fail. If creation of the process will exceed the overall process creation limit of the system, the process creation will fail.

PROCESS IDENTIFIERS (OR PID)

Every process has a unique process ID, a non-negative integer. Since the process ID is the only well-known identifier of a process that is always unique, it is often used as an identifier to guarantee uniqueness.

SPECIAL PROCESS IDS

Process ID 0 is the scheduler process and is often known as the swapper. No program on disk corresponds to this process. It is part of the kernel and is known as a system process.

Process ID 1 is the **init** process and is invoked by the kernel at the end of the bootstrap process. The program file for this program is */etc* **init**.

Process ID 2 is the page daemon. This process is responsible for supporting the paging of the virtual memory system. It's a kernel process.

PARENT PROCESS

Every process has got a parent process associated with it (although there are exceptions to this, which we'll discuss later). When a process is being created, it must be created by another process and this creating process is called the parent process, which equally has a unique ID.

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

CHILD PROCESS

When a process is being created to execute a program, this is usually viewed as a parent process.

Now, when this newly-created process goes on to create one or more processes, these processes are called child processes.

ORPHAN PROCESS

A process whose parent terminates is called an orphan process and is inherited by the **init** process.

ZOMBIE PROCESS

Once a process has been killed, the process turns itself into a zombie process. In this state, it is appropriately described as dead but not buried. This means that the process has no swapping image, but continues to exist in the process table, which gives it a dubious and not very useful status.

The zombie process can have its entry removed from the process table by its parent. The zombie can never be made executable again since it has no swapping image.

PROCESS TABLE

A process table is a fixed size array of structures where the control information about a process is stored. The size of this array places a fixed limit on the maximum number of processes that can exist at a time and can give rise to process table overflow errors.

PROCESS GROUP

In addition to having a process ID, each process also belongs to a process group. A process group is a collection of one or more processes. Each process group has a unique process group ID.

Process group IDs are similar to process IDs. The function **getpgrp()** returns the process group ID of the calling process.

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

PROCESS LEADER

Each process group can have a process group leader. The leader is identified by having its process group ID equal to its process ID.

PROCESS VERSUS PROGRAM

A process is a collection of identifiable entities within which a program is executed. The relationship between the process and program is not one-to-one because a process can execute more than one program at a time. It is also possible to remove a currently executing program from a process and start the execution of a new program in its place.

Foreground process

A foreground process is a process that runs from a terminal and ties up the terminal during its entire execution period. If the terminal is closed, the job will be killed.

Background process

A background process is a process that runs from a terminal but does not tie up the terminal during its entire execution period. If the terminal is closed, the job may or may not be killed.

If the **nohup** command is used to execute the job, the job will continue to execute even if the terminal is closed, otherwise the job will be killed. In the case of a job running with the **nohup** command, if the terminal is closed, the process is inherited by process ID 1 (known as the **init** process). A background process is started with an ampersand sign (&) at the end of the command.

PROCESS NICHE VALUE

Not all processes are created equal under Unix. Unix maintains a queue of processes ordered by priority.

Foreground processes, such as a user typing a command at a prompt, often receive a higher priority than background processes. However, one may want to run background processes at an even lower priority.

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

This is done using the command **nice**. The **nice** command modifies the scheduling priority of time-sharing processes. A process with a high **nice** number runs at a low priority, getting relatively little of the processor's attention. Similarly, jobs with a low **nice** number run at a high priority. When a command is issued with **nice** being mentioned, it is assigned a default priority that corresponds to a default **nice** value of 20. The **nice** values range from 0 to 20.

PROCESS CONTROLLING TERMINAL

The process controlling terminal is the terminal from which the process was started. In the **ps** listing, this is usually given as a tty or terminal ID. The **tty** command can be used to report to which 'terminal' you're currently connected. A controlling terminal is not required for a process to run. A background process always runs without a controlling terminal.

KILLING A PROCESS

A process can be killed using the **kill** command with a process number as an argument. For example:

```
kill 123
```

The **kill** command sends a particular **kill** signal to the specified process. For instance, it will send signal 15 (known as TERM) to the

Signal name	*Signal number*
TERM	15
KILL	9
TSTP	24
QUIT	3
HUP	1
INT	2

Figure 1: Signal name and signal number relationship

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

specified process in this example.

The **kill** command can be issued with a signal number (**kill -signal_number**) or signal name (**kill -signal_name**). The relationship between signal name and signal number is shown in Figure 1.

The command **kill -l** will list all the signal names.

However, a **kill** command may not always succeed in killing a process because the process in question can catch this signal and choose to ignore it. Therefore, a sure way to **kill** a process is to use signal_9.

When designing a program, it is a good idea and practice to design a signal handling routine that would process a caught signal in an appropriate manner. The program catches these signals usually to protect the execution of a critical section of the code.

PROCESS AND PROCESSOR

In a multiprocessor environment, a process can be bound to a specific processor, meaning that that processor will execute the process. When a process is not bound specifically to a processor, the scheduler will bind that process to a processor at random.

DAEMON PROCESS

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down, These processes are said to run in the background, because they do not have a controlling terminal.

Daemon process characteristics are:

- Runs with super privilege (a user ID of 0).
- Has no controlling terminals – the terminal name is set to question mark.
- Parented by the **init** process.

CODING RULES FOR DAEMON PROGRAMS

Coding rules for daemon programs are:

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

1 The first thing to do is call **fork** and have the parent exit. This does several things.

   If the daemon was started as a simple shell command, having the parent terminate makes the shell think the command is complete.

2 Call **setsid()** to create a new session.

3 Change the current working directory to the root directory.

4 Set the file mode creation mask to 0.

5 Close all unneeded file descriptors.

SAMPLE DAEMON PROGRAM

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
int   daemon-int (  void )
{
   pid-t         pid;
if  (  (  pid  =  fork ()  )  <  Ø  )
        return (-1);
else  if  (  pid  !=   Ø  )
       exit (Ø)   ;    /*  parent  exist  */
/*  child  continues   */
setsid();          /*  become  session  leader          */
chdir ("/" );      /*  change  working  directory       */
umask (Ø);         /*  clear  file  mode  creation  mask  */
return (Ø) ;
}
```

Arif Zaman
Analyst/Programmer
High-Tech Software (UK)
© Xephon 2002

Introduction to the shell

The shell is the outermost layer of the operating system and is actually a programming language that is used to control processes and files, as well as to start and control other programs. The shell manages the interaction between the user and the operating system by prompting

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

the user for input. It interprets that input for the operating system, and then handles any resulting output. The shell runs from the moment you log in until you log off.

Normally, you interact with the shell by entering a command, and wait for the shell to respond by displaying a new prompt before you enter the next command.

It is also possible to put a sequence of commands in a file and have the shell run these commands one after the other. When you use the shell to run commands from a file, you are using it in a non-interactive way. The shell will read commands from the file, rather than prompt you and read what you type.

A file that contains a sequence of commands is called a shell program or shell script. The shell has a number of features that make it similar to high-level programming languages. These include:

- Variables.
- Flow control constructs, such as *if-then* statements and loops.
- Signal handling.

These features make it possible to create shell programs that consist of more than just a list of simple commands. Shell scripts can provide an easy means of carrying out tedious tasks, large or complicated sequences of commands, and routine tasks, and, by following certain rules, the command sequences in a script can be carried out simply by typing the name of the shell script file.

SHELLS AVAILABLE

Several shells are available to interface with AIX, and a particular shell is chosen by invoking the command with which it is associated. Although the shell plays a special role in the operating system, it is really just a program, and can be run like any other program.

AIX Version 4 is provided with the following shells:

- Korn shell (invoked with the **ksh** command). This is the standard shell that appears when you log in. It is more sophisticated than

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

the Bourne and C shells and incorporates many of their features, plus other enhancements.

- Bourne shell (invoked with the **bsh** or **sh** command).
- Restricted shell (a subset of the Bourne shell, invoked with the **Rsh** command). It is used as a security feature to limit user access to a subset of AIX commands and is not used in shell programming.
- C shell (invoked with the **csh** command).
- Trusted shell (invoked with the **tsh** command). It is an enhanced security shell not used in shell programming.
- Remote shell (invoked with the **rsh** command). Normally used for logging on to remote hosts and is used in shell programming only to remotely execute a command.

COMMAND TERMINATORS AND PIPELINES

In shell scripts you can put several commands on the same line by separating them with any of the command terminators. There are no hard and fast rules and it is purely a matter of personal preference.

Since the new-line character is normally treated as a command terminator, you must do something special when you want to break up a command that is too long to fit on a single line; for example you may want to split the command so that your script is easier to read. To do this, type \ just before you press <CR>, and then continue typing the command. A new-line character that is preceded by a \ is said to be 'escaped'.

In much the same way, you can place a pipe symbol at the end of a line and continue the pipeline on the next line if you want to make the script easier to read.

Tonto Kowalski
Guru (UAE)
© Xephon 2002

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

AIX 5L routing enhancements

In AIX 5L (5.1) several new features were added to the TCP/IP stack and some enhancements were made. Of these, perhaps the ones that will be of the greatest use to the majority of the AIX community are the enhancements to routing, specifically multipath routing (sometimes referred to as equal cost multipath routing), and dead gateway detection. This article explains the concepts of these, and demonstrates their usage in making your routing table more efficient and robust.

MULTIPATH ROUTING

An example network diagram is shown in Figure 1.

In this example our AIX box has a Token Ring interface 11.1.1.1, and an Ethernet interface 10.1.1.1, both of which have a subnet mask of 255.255.255.0. The default route goes via the Ethernet interface to 10.1.1.2. The current routing table is shown below:

```
Routing tables
Destination      Gateway        Flags  Refs    Use  If   PMTU Exp Groups
Route Tree for Protocol Family 2 (Internet):
default          10.1.1.2       UG      0       89  en0     -   -
10.1.1/24        10.1.1.1       U       4     2435  en0     -   -
11.1.1/24        11.1.1.1       U       4    12719  tr0     -   -
127/8            127.0.0.1      U       6     9674  lo0     -   -

Route Tree for Protocol Family 24 (Internet v6):
::1              ::1            UH      0        0  lo0 16896   -
```

Note: to retain the simplicity of the routing table, Path MTU discovery has been turned off (see **no** options **tcp_pmtu_discover** and **udp_pmtu_discover**).

But, with two possible routes to the default router, it would be nice if we could balance the load between the Token Ring and Ethernet networks. The question is, how?

One possible way of sharing the load between the two adapters is to add specific host routes or network routes to manually route to certain destinations via a certain interface. In the network diagram in

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

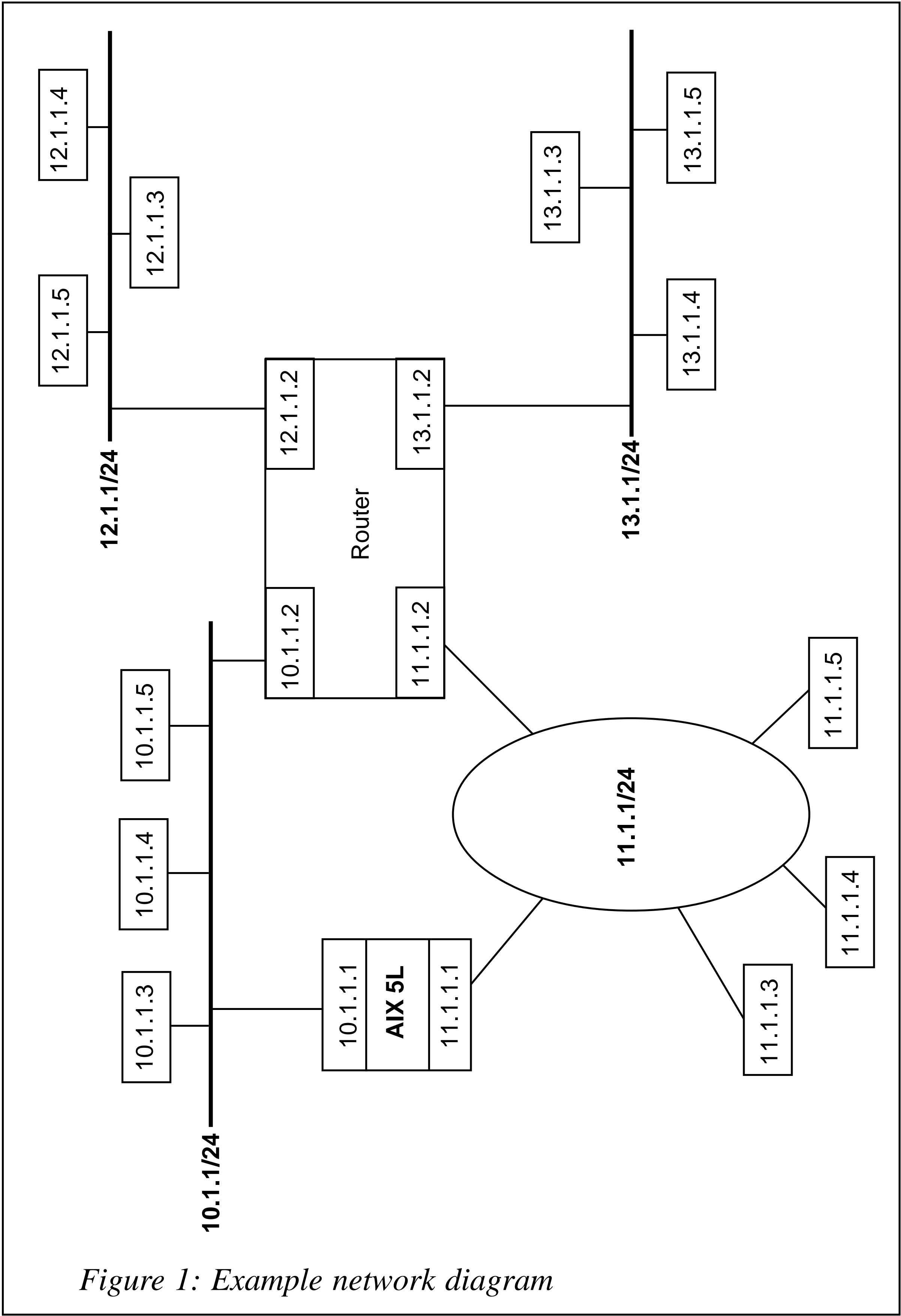

Figure 1: Example network diagram

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

Figure 1 it would be reasonably easy to use the Ethernet adapter to route to the 12.1.1/24 network and the Token Ring adapter to route to the 13.1.1/24 network; it would just be a case of adding two network routes, which would give us a few extra lines of **netstat -rn** output:

```
12.1.1/24          10.1.1.2        UG       0         0  en0       -   -
13.1.1/24          11.1.1.2        UG       0         0  tr0       -   -
```

In the case of this example network, this is an easy split, but there are still problems – for example, what if 90% of all traffic goes over the 12.1.1/24 network; suddenly we're not balancing equally. Again we could intervene manually by adding host routes via certain adapters instead of network routes. In which case we'd probably end up with something along the lines of:

```
12.1.1.3           10.1.1.2        UGH      0         0  en0       -   -
12.1.1.4           11.1.1.2        UGH      0         0  tr0       -   -
12.1.1.5           10.1.1.2        UGH      0         0  en0       -   -
13.1.1.3           11.1.1.2        UGH      0         0  tr0       -   -
13.1.1.4           10.1.1.2        UGH      0         0  en0       -   -
13.1.1.5           11.1.1.2        UGH      0         0  tr0       -   -
```

This approach takes a lot of effort, even on a small network. With a medium-sized network of only 100 hosts of so, this would just take too much work to manage properly, and the routing table would be too unruly to work with easily.

Perhaps these are the reasons why most system managers don't actually make this effort in trying to balance the load between the two adapters, and instead just use a default route. This was the story in AIX 4.3 and versions previous to that, but now, in AIX 5L, things have changed.

Prior to AIX 5L a route had to be unique, identified by its destination and netmask. With multipath routing this criterion is no longer used. We are now able to add multiple routes to the same destination, which AIX then routes through based on a 'round-robin' method.

The simplest implementation we could use would be to simply add a second default route via our tr0 adapter:

```
# route add 0 11.1.1.2
11.1.1.2 net 0: gateway 11.1.1.2
```

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

```
# netstat -rn
Routing tables
Destination      Gateway         Flags  Refs     Use  If   PMTU Exp Groups
Route Tree for Protocol Family 2 (Internet):
default          11.1.1.2        UG      0        89  tr0     -   -    =>
default          10.1.1.2        UG      0        89  en0     -   -
10.1.1/24        10.1.1.1        U       4      2435  en0     -   -
11.1.1/24        11.1.1.1        U       4     12719  tr0     -   -
127/8            127.0.0.1       U       6      9674  lo0     -   -
Route Tree for Protocol Family 24 (Internet v6):
::1              ::1             UH      0         0  lo0 16896   -
```

The '=>' symbol tells us that this route is a duplicate. We can test whether AIX is routing on a round-robin-basis using something simple such as **traceroute**, eg:

```
# traceroute 12.1.1.3
trying to get source for 12.1.1.3
source should be 11.1.1.1
traceroute to 12.1.1.3 (12.1.1.3) from 11.1.1.1 (11.1.1.1), 30 hops max
outgoing MTU = 1492
 1  11.1.1.2 (11.1.1.2)  10 ms  4 ms  3 ms
 2  12.1.1.3 (12.1.1.3)   7 ms  7 ms  7 ms
```

then run it again (note the different source address used):

```
# traceroute 12.1.1.3
trying to get source for 12.1.1.3
source should be 10.1.1.1
traceroute to 12.1.1.3 (12.1.1.3) from 10.1.1.1 (10.1.1.1), 30 hops max
outgoing MTU = 1500
 1  10.1.1.2 (11.1.1.2)  10 ms  4 ms  3 ms
 2  12.1.1.3 (12.1.1.3)  7 ms  7 ms  7 ms
```

One important point to note about multipath routing is that it cannot be used in conjunction with path mtu discovery (PMTU), so this must be turned off beforehand.

A common question that came up from people running 4.3 is, could they balance the load by configuring two interfaces in the same subnet? This not only didn't work, but was almost completely useless. In 4.3 only the first interface that came up would get the 'interface' route – identified by a single U in the flags.

Let's look at an example:

```
# netstat -in
Name  Mtu   Network    Address           Ipkts Ierrs    Opkts Oerrs  Coll
tr0   1492  link#3     0.4.ac.64.f3.de   1069     0        6     0     0
```

 © 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

```
tr0   1492  10.10.10   10.10.10.1        1069     0       6     0     0
tr1   1492  link#4     0.4.ac.64.a3.a0    215     0      87     0     0
tr1   1492  10.10.10   10.10.10.2         215     0      87     0     0
lo0   16896 link#1                        198     0     202     0     0
lo0   16896 127        127.0.0.1          198     0     202     0     0
lo0   16896 ::1                           198     0     202     0     0
# netstat -rn
Routing tables
Destination      Gateway        Flags  Refs    Use  If   PMTU Exp Groups
Route Tree for Protocol Family 2 (Internet):
10.10.10/24      10.10.10.1     U      2       10  tr0      -   -
127/8            127.0.0.1      U      6      188  lo0      -   -
Route Tree for Protocol Family 24 (Internet v6):
::1              ::1            UH     0        0  lo0 16896   -
```

Notice in this example there is no reference to tr1 in the **netstat -rn** output, because adding another interface route on tr1 would cause a duplication of routes, which 4.3 did not support. More crucially, if traffic came in for tr1, it would be returned over the tr0 interface – thus rendering tr1 redundant.

In 5.1 this has been resolved: multiple interfaces in the same subnet are supported, and multipath routing will load-balance between them. Using the same **netstat -in** as before, our **netstat -rn** would now look like:

```
Routing tables
Destination      Gateway        Flags  Refs    Use  If   PMTU Exp Groups
Route Tree for Protocol Family 2 (Internet):
default          11.1.1.2       UG     0        89  tr0      -   -    =>
default          10.1.1.2       UG     0        89  en0      -   -
10.1.1/24        10.1.1.1       U      4      2435  en0      -   -
11.1.1/24        11.1.1.1       U      4     12719  tr0      -   -    =>
11.1.1/24        11.1.1.4       U      2        10  tr1      -   -
127/8            127.0.0.1      U      6      9674  lo0      -   -
Route Tree for Protocol Family 24 (Internet v6):
::1              ::1            UH     0         0  lo0 16896   -
```

This would load-balance between tr0 and tr1 on the 11.1.1 subnet, but only over tr0 and en0 from the default route. So in order to additionally balance our default route using tr1 we need to add another default route.

Our problem here is that our normal command to add the default route:

```
# route add 0 11.1.1.2
```

would add the route over the tr0 interface – and we already have that route. Therefore a new flag has been added to the route command to let us specify which interface we want the route to use. This is the **-if** flag. We'd add the route in question with:

```
# route add Ø 11.1.1.2 -if tr1
```

resulting in the routing table:

```
Routing tables
Destination      Gateway        Flags  Refs    Use  If   PMTU Exp Groups
Route Tree for Protocol Family 2 (Internet):
default          11.1.1.2       UG       Ø       89  trØ     -   -    =>
default          11.1.1.2       UG       Ø       12  tr1     -   -    =>
default          1Ø.1.1.2       UG       Ø       89  enØ     -   -
1Ø.1.1/24        1Ø.1.1.1       U        4     2435  enØ     -   -
11.1.1/24        11.1.1.1       U        4    12719  trØ     -   -    =>
11.1.1/24        11.1.1.4       U        2       1Ø  tr1     -   -
127/8            127.Ø.Ø.1      U        6     9674  loØ     -   -
Route Tree for Protocol Family 24 (Internet v6):
::1              ::1               UH       Ø       Ø  loØ 16896   -
```

Now we're balancing our load over three adapters. The next question is, what happens if one of our next-hop routers goes down? This brings us neatly on to our next topic, *Dead Gateway Detection*.

DEAD GATEWAY DETECTION

Dead Gateway Detection (DGD), as the name would suggest, is designed to detect whether one of your gateways is dead. It does this in a few different ways, and the default is *off*.

The idea of DGD working out whether your gateway is alive or dead is, in itself, fairly useless. The key feature is that if you've got another route to the same destination (ie you're using multipath routing) then it will switch all the traffic over to the alternative route. The two methods of detecting whether the gateway is dead are through passive dead gateway detection and active dead gateway detection.

Passive Dead Gateway Detection (PDGD) is either on or off, and affects all of the routes in the table. It's turned on via the **no** command:

```
# no -a | grep passive_dgd
     passive_dgd = Ø

# no -o passive_dgd=1
```


© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

PDGD relies on hints from the other communication layers to determine whether a gateway is alive or not, such as excessive TCP retransmissions, or getting no response from ARP attempts.

However, if the protocols in use can't give hints about the gateway (for example a UDP connection) then it will not be possible to determine whether the gateway is alive or dead. This means that PDGD can be quite slow to detect that, in fact, a gateway is dead, or that a dead gateway has come back up again. One advantage of this method though is that it requires very little additional overhead, and so is recommended for machines that are deemed 'non-critical'.

Active Dead Gateway Detection (ADGD), as the name suggests, actively goes out and tests whether the gateways are up or down, by the simple means of pinging them. Although there is no official set method of performing Dead Gateway Detection, the recommendations in RFCs 1122 and 816 advise that pinging should not be used, or kept to an absolute minimum, so in choosing this method AIX is not fully compliant with the RFC.

ADGD is done on a per route basis, and is activated by another additional flag for the route command, **-active_dgd**. Routes which have ADGD set are visible by the new 'A' flag in the **netstat -rn** listing.

As an example, we can delete one of our default routes, and re-add it with the **active_dgd** flag:

```
# route delete Ø 11.1.1.2 -if tr1
tr1 net Ø: gateway 11.1.1.2
# route add Ø 11.1.1.2 -if tr1 -active_dgd
-active_dgd net Ø: gateway 11.1.1.2

# netstat -rn
Routing tables
Destination      Gateway        Flags  Refs    Use  If   PMTU Exp Groups
Route Tree for Protocol Family 2 (Internet):
default          11.1.1.2       UG      Ø       89  trØ     -   -    =>
default          11.1.1.2       UGA     Ø        Ø  tr1     -   -    =>
default          1Ø.1.1.2       UG      Ø       89  enØ     -   -
1Ø.1.1/24        1Ø.1.1.1       U       4     2435  enØ     -   -
11.1.1/24        11.1.1.1       U       4    12719  trØ     -   -    =>
11.1.1/24        11.1.1.4       U       2       1Ø  tr1     -   -
127/8            127.Ø.Ø.1      U       6     9674  loØ     -   -
Route Tree for Protocol Family 24 (Internet v6):
::1              ::1            UH      Ø        Ø  loØ 16896   -
```

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

As already discussed, ADGD works by simply pinging the routers. By default a ping will occur every five seconds to every route that has ADGD enabled on it. If after three successive pings it fails to get a response, the router is deemed dead.

Before looking at how AIX interprets this, it's important to think about how much extra traffic this pinging will generate. On a single machine, pinging its default router, the answer is not a lot. With a large network of AIX boxes with lots of ADGD-enabled routes, we could get quite a ping-storm going, with a noticeable impact on network performance. Because of this potential hit in network performance, plan carefully which machines and routes need ADGD, or just use it on machines that are deemed 'critical'. AGDG and PDGD aren't exclusive to one another, so ADGD can be used on 'critical' routes, while PDGD can be used on the others.

Helpfully enough, pinging every three seconds is only the default value, and the DGD options can be tuned with the **no** options:

```
# no -a | grep dgd
     dgd_packets_lost = 3
     dgd_retry_time = 5
     dgd_ping_time = 5
     passive_dgd = 0
```

Let's look at these in detail:

- dgd_packets_lost are the number of pings that aren't responded to for the route to be thought of as dead.
- dgd_retry_time is the number of minutes before the route is put back to its normal state (if it's still down it'll fail over again).
- dgd_ping_time is the number of seconds between each ping.
- Passive_dgd, as discussed, if set to 1, will turn on PDGD.

In both PDGD and ADGD, the way in which AIX interprets a route as being dead is the same. In AIX 5L, each route also has a cost attached to it (this will be familiar to anyone who's worked with RIP-2 or OSPF). AIX will simply then take the lowest-cost route possible to the required destination.

If there's only one possible route to the destination in question, then adding a cost to the route will make no difference – if there's only one

 © 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

route to the destination, then it's always going to have the lowest cost. If we have multiple routes to the same destination, as in the previous examples, then changing the costs of certain routes will have a marked result. If not specified, routes will have a cost of 0 – use the **-hopcount** flag to assign a route. In order to see the costs assigned to the routes, the 'C' flag is added to the **netstat -rn**.

As an example, we'll delete another one of our default routes, and then re-add it with a cost of 16:

```
# route delete 0 10.1.1.2 -if en0
en0 net 0: gateway 10.1.1.2
# route add 0 10.1.1.2 -if en0 -hopcount 16
3 net 0: gateway 10.1.1.2
# netstat -rnC
Routing tables
Destination    Gateway         Flags   Refs     Use  If   Cost Config_Cost
Route Tree for Protocol Family 2 (Internet):
default        11.1.1.2         UG        0        0  tr0     0          0
default        11.1.1.2         UGA       0      243  tr1     0          0
default        10.1.1.2         UG        0       89  en0    16         16
10.1.1/24      10.1.1.1         U         4     2435  en0     0          0
11.1.1/24      11.1.1.1         U         4    12719  tr0     0          0
11.1.1/24      11.1.1.4         U         2       10  tr1     0          0
127/8          127.0.0.1        U         6     9674  lo0     0          0
Route Tree for Protocol Family 24 (Internet v6):
::1            ::1              UH        0        0  lo0     0          0
```

When a dead route has been detected, the kernel assigns the highest cost possible to the route, which will show as MAX (this is equivalent to a cost of 2,147,483,647) in the **netstat -rnC** output. When choosing the lowest-cost route, obviously the kernel is going to avoid the MAX cost routes and choose the lowest cost options.

We've got a very obvious flaw in our current routing table: we have set ADGD on the default route on tr1. But if the 11.1.1.2 router goes down then we would still use the route to 11.1.1.2 on tr0!

As an example, we'll set ADGD on the default route on tr0 as well and then take the 11.1.1.2 router down. After a matter of seconds, our resulting routing table looks like:

```
# netstat -rnC
Routing tables
Destination    Gateway         Flags   Refs     Use  If   Cost Config_Cost
Route Tree for Protocol Family 2 (Internet):
default        11.1.1.2         UGA       0       14  tr0   MAX        MAX
```

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

```
default        11.1.1.2        UGA        Ø       243  tr1   MAX        MAX
default        1Ø.1.1.2        UG         Ø       495  enØ    16         16
1Ø.1.1/24      1Ø.1.1.1        U          4      2435  enØ     Ø          Ø
11.1.1/24      11.1.1.1        U          4     12719  trØ     Ø          Ø
11.1.1/24      11.1.1.4        U          2        1Ø  tr1     Ø          Ø
127/8          127.Ø.Ø.1       U          6      9674  loØ     Ø          Ø
Route Tree for Protocol Family 24 (Internet v6):
::1            ::1             UH         Ø         Ø  loØ     Ø          Ø
```

So all traffic needing the default route will go out of en0 on to the router 10.1.1.2. In five minutes' time, the routes will be given their original costs again, and the ping will be retried to see if they are still dead.

Dead gateway detection and multipath routing offer us some useful features; not only the obvious functions of having multiple routes to the same destination for load balancing, and detecting that a gateway is down, but by combining these we can configure some routes or interfaces to be used only in exceptional circumstances.

This is a new function to AIX 5L and there were a few defects in the initial release. Although these have been addressed, it's recommended running on the latest maintenance level.

(UK) © Xephon 2002

Shell-scripting tricks

SHELL INTRODUCTION

If you have any command line interaction with your Unix server, the chances are that it involves a shell somewhere – such things as sh, bash, csh, ksh, and the less frequently encountered zsh, scsh, and es.

All these are different packages that handle the commands you type, spawning processes, and passing data back and forth between them. They differ in such areas as key combinations, behaviour (and set) of output redirector syntaxes, variable editing, which platforms come with which shells by default, and so on.

This article provides the backing terminology to take you from simply

 © 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

entering commands to constructing longer and more complex instructions, by understanding the building blocks behind shell scripting. Throughout this document, the *bash* shell is used; the vast majority of examples presented will also work in vanilla Bourne *sh*; bash's open-source nature means that it is freely available, and indeed it is making its presence felt at many installations.

SHELL LOOK AND FEEL

The shell is responsible for handling how commands look, and the editor keypresses that can be used to generate them. Generally, there are two 'modes' for a shell – it can either be in edit mode, which resembles vi, or it uses a readline and feels like a one-line version of emacs. Modern shells seem to default to the latter; they all have the option to rebind the keys, with either:

```
set -o vi
```

in ksh and bash, or:

```
bindkey -v
```

in [t]csh.

The most useful keys to know about are probably the following (readline versions presented first):

- Cursor left/right will move back and forth along the line a character at a time. In vi-mode, use *Esc* to get into command mode and use h and l correspondingly.
- Cursor-up/down will cycle through the command history one line at a time. In vi-mode, use *Esc* to get into command mode and use k and j respectively.
- ^A, ^E will take you to the start or end of a line. In vi-mode, use Esc and then I (to jump back into insert-at-start-of-line) or *Esc*+| or ^ to move to the start of line. Use $ to move to the end, and A to append after the end of the line.
- M-f, M-v will move you forwards or backwards a word. In vi-mode, Esc to enter command mode, then either W or w, or B or b, to move forwards and backwards by (big or small) words.

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

- ^R will start interactively asking for letters out of previous commands in the history, so you can press ‘l s’ for the last command including those letters, and then press ^R again for the previous match, or add more characters. (Use a cursor-key or ^E to bring this command to the fore.)

SHELL VARIABLES

All shells support the idea of having both local and environment variables. They are set as follows:

- set var=value (csh)
- var=value (sh/bash)
- setenv var value (csh)
- export var (sh/bash).

The difference is that the (current) shell alone will see shell variables; however, environment variables are inherited down the process tree.

For example:

```
init(1)-+-apache(329)-+-apache(3440)
        |             |-apache(3441)
        |             |-apache(3442)
        |             |-apache(3443)
        |             |-apache(3444)
        |             '-apache(3463)
        |-boa(238)
```

is the process tree of my apache and boa Web servers; I invoked this from /etc/init.d/apache with the LD_PRELOAD variable set to include ‘/lib/libsafe.so.1’. Each of the child processes now also has that environment variable, but boa has not.

The presence of a shell or environment variable is determined by typing ‘set’ on its own. The presence of an environment variable only is determined by running the ‘env’ command. This is an important difference; if you were to debug a situation where one command saw a variable but something that the first command spawns does not see it, then typing ‘set’ to answer ‘is the environment variable FOO set?’ would be a logical error. Use env for this purpose. It is also conventional

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

(unlike in the above) for environmental variables to be written in upper case.

There is also an ‘unset’ feature to all these shells that works both for shell and environment variables. Bash, kshm, and zsh also support shortening ‘FOO=value; export FOO’ into the more succinct:

```
export FOO=value
```

Shells typically reserve certain variable names for their own purposes. These include:

- $$ – the pid of the current shell process
- $? – the exit status of the last command: typically 0 means it exited successfully, non-zero indicates an error.
- $1..$9 – when writing a shell script, these are the parameters passed on the command line.
- $* – when writing a shell script, this is all the parameters on the command line.
- IGNOREEOF – if you send the shell an EOF signal (by pressing ^D), then, with this variable set, it will say “Type exit to log out”; with it unset, it will log out.
- $() – surprise! This isn’t a variable; see below under *backticks*.
- $(()) – surprise! This isn’t a variable; some shells (notably bash/ zsh) use it to denote arithmetic operations.

JOB CONTROL

One of the features of the shell is to provide control over the processes you spawn.

When you tell the machine to run ‘ls’, for example, the shell receives your input (the command), searches a path for the ‘ls’ executable, forks, executes the command, and ensures that output comes to your current tty.

Straight away we meet the first differences between the various shells: the path-searching mechanism is through an environment variable

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

called ‘PATH’ in sh, bash, and ksh; in csh, tcsh, and zsh, it’s done through an array variable called *path*, and you have to run ‘rehash’ in order for the shell to update its hash of what commands are available where in the path.

In the case of something as simple as ‘ls’, obviously no special job-control is required. However, if you had a shell script that took half a day to run, the last thing you would want is to tie up your terminal window waiting for the output. This is where all the shells support the following:

- Use of & to background a process – output still comes to this tty.
- ^C (technically, whatever ‘stty intr’ is set to) to terminate the currently running process.
- ^Z to suspend the currently running process.
- A ‘jobs’ command to show what processes are currently running from this shell.
- ‘bg’ and ‘fg’ to take the currently suspended process and resume running in the background, or to take a backgrounded process and bring it to the foreground.

Hence, the following:

```
bash$ sleep 120
^Z
[1]+  Stopped                 sleep 120
bash$ bg
[1]+ sleep 120 &
bash$ jobs
[1]+  Running                 sleep 120 &
bash$ sleep 240 &
[2] 1281
bash$ jobs
[1]-  Running                 sleep 120 &
[2]+  Running                 sleep 240 &
bash$
```

shows the bash shell starting a sleep process for two minutes, that process being backgrounded, a second sleep being started for four minutes already in the background, and two listings of the jobs present in that shell instance.

Normally, any one of these jobs can be brought to the fore by typing:

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

```
fg %n
```

for some suitable *n*; in some shells, it can be abbreviated to just '%n'. You can also terminate a particular one by using 'kill %n'.

OUTPUT REDIRECTION

Shells do a lot of parsing of the command line before the command(s) being executed actually see it. The most obvious of these is the stdin/stdout redirection; a command's output can be sent to another file, or appended to it, or piped to another command; its input, by the same token, can be from a file or a pipe on the left.

For example:

```
bash$ ls
CVS        shell.aux   shell.html~   shell.log  shell.tex
Makefile   shell.dvi   shell.latte   shell.pdf  shell.txt
Makefile~  shell.html  shell.latte~  shell.ps
bash$ ls -1
CVS
Makefile
Makefile~
shell.aux
shell.dvi
shell.html
shell.html~
shell.latte
shell.latte~
shell.log
shell.pdf
shell.ps
shell.tex
shell.txt
bash$ ls -1 | wc -w
     14
bash$ ls -1 | wc -w > nolines
bash$
```

Here we have four commands executed, showing the first phases of how to build up a lengthy command line. The essential thing to watch is the nature of the data at each stage: when we execute 'ls', the command runs, and produces one layout of data (multi-columned). When we change to 'ls -1', the data is just one column wide. Then we feed that column of data into the **wc** command, and it produces one (indented) number, the number of words in the input stream (which came in from the pipe). Finally, we take that one number of output,

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

and either create a new file, or truncate any existing one, called ‘nolines’, with just that number as its contents.

It is also worthwhile noting that the command line that the ‘ls’ process sees is only ‘ls -1’; the command line that ‘wc’ sees is ‘wc -w’. It is the shell itself that splits the command line into these three components, spawns the processes in the right order, and joins the stdout of one to the stdin of the next, finally putting the output directly into a file.

Presumably there was some ulterior motive for wanting to do this.

The redirection operators

There are five redirection operators of interest, all of which appear after a given command on the command line:

- > – creates a new file, or truncates an existing one, putting stdout into that file.
- >> – appends to an existing file, or creates a new one if necessary, putting stdout into that file.
- < – links the process’s stdin from a file of data.
- << – used to describe ‘here-documents’, where everything from ‘<<EOF’ until a line containing ‘EOF’ on its own is read and parsed as stdin to the command.
- | – used to pipe the output from the command on the left to the command on the right.

Noclobber

The behaviour of > and >> above is shell and option-dependent. In csh on Solaris, for example, the default used to be to have ‘set noclobber’ set, meaning that > required there to be no file of the destination name present, and that >> required the file to exist before appending to it. It is possible to invert this behaviour by appending a ! to each operator, eg >>! for ‘append or create as necessary’. However, it would also be sane to ‘unset noclobber’ in your ~/.cshrc.

Streams

Mention has been made above of ‘stdin’ and ‘stdout’. C programmers

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

will know already what they are: streams of data, similar to file handles, where stdin is the link to the input (tty), stdout is your process's link to the output (tty), and stderr is a link to the error device (probably also a tty). The idea is that any utility behaving as a filter (eg 'wc' in the above) will read stdin, put primary output on stdout, and report any errors it encounters to stderr.

Needless to say, your shell can also handle stderr: the canonical way to reference it is with '2>' or '2|'. It can also be bundled with stdout so that both streams can be processed by the same command later.

For example:

```
bash$ ls eek
ls: eek: No such file or directory
bash$ ls eek > /dev/null
ls: eek: No such file or directory
bash$ ls eek 2> /dev/null
bash$
```

The first two lines show stderr not being handled at all, and therefore still going to the terminal; in the third line, the same 'error' occurs, but it is explicitly handled and ignored.

The various shells differ regarding how they go about joining the various streams, but commonly either '|&' and '>&' and '>>&' (the csh variants) or 'command > file 2>&1' or 'command | command 2>&1' (sh/bash variants) are supported. Typically, zsh supports both.

ESCAPING FROM THE SHELL

As mentioned above, the shell does a lot of interpretation of the command lines you give it. This revolves around a set of shell-specific characters: not only the redirectors above, but also the !, $, ", ', (), and even the humble space character, are all parsed specially.

If you need to pass these characters to a shell command, you have to escape it from the shell itself. For one character, it suffices to prepend a single \ character in front of it. For something longer, you can choose between "" around the string, in which case any variables present between the "" will be expanded, and '', in which case variable references will not be expanded.

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

USEFUL COMMANDS

There are several commands of use for manipulating the data as it flows through a command pipe. In particular, commonly used commands and options include:

- ls – produces a list of files in the current directory. Notable options include -F to append a character defining the file type, -C to force multi-column display, -1 for single column, -l for a long (detailed) listing, -t for output sorted by time, -S for output sorted by size, -r for reversed sort, -h for sizes displayed in sensible units.
- echo – just write out a string.
- cat – simply dumps the contents of the file to screen. With -s, reduces multiple blank lines down to one; with -v, displays hex representations of binary characters.
- uniq – eliminates consecutive duplicate lines; with -c, gives a count of how many of each line there were.
- head and tail – prints the first (last) 10 lines (by default; use -N for some other number) of either the file/s given, or stdin, to stdout. Useful for sampling data without having to watch a potentially large file go flying by on-screen.
- find – this is a more complex one; takes a list of start directories or files, applies a set of predicates to each file found, and performs an action on each. For example:

  ```
  find / -type f -xdev -exec cat {} \; > /dev/null
  ```

  will find all files on the same partition as /, and cat them one after the other, with results going into /dev/null. (This is a good way to add 1 to the system load for stress purposes.)
- grep – searches a set of files or stdin for a regular expression (see regex(5) or similar document on your system); by default the whole line matching the regex is printed. For example,

  ```
  ps -ef | grep tim
  ```

  will first produce a list of all the processes on the box, and then display all those that contain the string ‘tim’ somewhere. This is

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

crudely used to find a user's processes; learn how to use your system's **ps** command properly if that's what you really want.

- cut – remove all but a range of fields or columns from a line. Option -d' ' will set space as the delimiter, and -f will specify a field range. For example, see:

```
bash$ ls -1 | cut -c6-

ile
es
.aux
.dvi
.html
.latte
.latte~
.log
.pdf
.ps
.tex
.txt
bash$
```

This shows cut displaying only the sixth onwards fields. The first line of output was from 'CVS', which doesn't have that many characters, so is blank.

- sed – stream editor. Its main use is for performing regex-based search & replace commands on each line of input, eg:

```
bash$ ls
CVS       nolines    shell.dvi   shell.latte   shell.log  shell.ps
          shell.txt
Makefile  shell.aux  shell.html  shell.latte~  shell.pdf  shell.tex
bash$ rm nolines
bash$ ls -1 | sed 's/shell/foo/g'
CVS
Makefile
foo.aux
foo.dvi
foo.html
foo.latte
foo.latte~
foo.log
foo.pdf
foo.ps
foo.tex
foo.txt
```

shows all the references to 'shell' becoming 'foo' instead.

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

- awk is used for field manipulation; an awk script is a series of condition-expression pairs where the expression is evaluated when the condition is true. For example,

```
bash$ netstat -an | awk -F: '/^udp/ print $1'
udp        0      0 0.0.0.0
udp        0      0 0.0.0.0
udp        0      0 195.149.39.120
udp        0      0 127.0.0.1
udp        0      0 0.0.0.0
udp        0      0 0.0.0.0
```

  shows awk splitting the output from **netstat** based on ':', and then whether the line starts with the string 'udp' or not determines whether the first column is reprinted.

EXAMPLES FROM SCRATCH

Support fun

It's 11pm when your pager alerts you to the fact that connections to your Web server are intermittent. You log on to the server, and type **netstat -an** to see all the socket states. However, the output flies by for so long, you have to ^C it; you don't get any picture whether the problem is that there are too many sockets in ESTABLISHED state (ie the Web server is misconfigured), in SYN_RECV state (the Web server is not processing them fast enough), or in SYN_SENT state (because the Web server proxy-passes certain requests backwards, and the back-end server could be down).

Obviously, it would be nice to produce a small two-column table showing how many sockets there are in each status; that way, peeling off the above information would be trivial. The question is, how do you get that data?

We see from the **netstat** command that we can obtain a single line per socket with the status; we just need to remove surplus data and calculate how many of each distinct type there are. Working from left to right, we have:

```
bash$ netstat -ant  | head -6
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address     Foreign Address         State
tcp        0      0 0.0.0.0:993       0.0.0.0:*               LISTEN
```


© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

```
tcp        0      0 0.0.0.0:995        0.0.0.0:*              LISTEN
tcp        0      0 0.0.0.0:10022      0.0.0.0:*              LISTEN
tcp        0      0 0.0.0.0:110        0.0.0.0:*              LISTEN
tcp        0      0 127.0.0.1:143      0.0.0.0:*              LISTEN
tcp        0      0 127.0.0.1:80       0.0.0.0:*              LISTEN
```

(This shows only the first six lines of output, as a sampler.) We're interested in only the last column of each line. We know that **cut** would not make it easy to retrieve the last column unless we experimented with the column position at which it started; however, **awk** has exactly this facility, along with the ability to split on multiple consecutive instances of the separator, by default:

```
bash$ netstat -ant  | awk 'print $NF' | head
established)
State
LISTEN
LISTEN
LISTEN
LISTEN
LISTEN
LISTEN
...
```

Now we have exactly the sort of data stream that lends itself to being counted, so we're looking at seeing how many of each there are. If we sort the list and then run it through 'uniq -c', we'll get the following:

```
bash$ netstat -ant  | awk 'print $NF' | sort | uniq -c
     11 ESTABLISHED
     15 LISTEN
      1 State
      2 TIME_WAIT
      1 established)
```

As a last refinement, the output wants sorting again numerically:

```
bash$ netstat -ant  | awk 'print $NF' | sort | uniq -c | sort -n
      1 FIN_WAIT2
      1 State
      1 established)
      2 CLOSE_WAIT
      7 TIME_WAIT
     11 ESTABLISHED
     15 LISTEN
bash$
```

Voila! we have exactly the information required to debug the Web server's problems.

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

Sysadmin fun

You've just installed a clean build of the latest version of your favourite OS and are in the process of restoring your users' data across to the new box when you find that all the users' files are owned by uid and gids, instead of usernames/groups. Unfortunately, the new version of your OS has also edited /etc/passwd,group,shadow so that there is a new uid for your httpd and various other services, so you can't just blindly copy across the old passwd, group, shadow files.

There are two ways in which to fix this, depending on whether your users' passwords are predictable or not. If so, you notice that there are two directories where every user is listed: /home, and /var/spool/mail. It doesn't really matter which of these you choose, but I'd recommend the latter (assuming everyone has mbox delivery) on the grounds that you don't have subdirectories to confuse the issue.

Here there are another two ways to fix things: if you're on an older Unix system without the GNU tools, you have to manipulate 'ls -l' output, something like this:

```
bash$ ls -l
total 6336
-rw-rw——    1 501   mail      6487065 Dec 11 19:54 tim
bash$ ls -l | awk 'NF>3 print "adduser -u", $3, $NF'
adduser -u 501 tim
bash$
```

This works by taking the output from 'ls -l', and printing a string composed of an 'adduser' command and various fields from the ls -l output. There is a condition applied, that the number of fields (NF, in awk) is >3 before it prints the combination of fields.

This can then be piped through 'sh' and all the output commands will be executed.

The alternative, for those with GNU **find**, is:

```
bash$ ls -l
total 6744
-rw-rw——    1 501   mail      6905205 Dec 11 20:16 tim
bash$ find * -type f ! -name root -printf adduser %u %p\n | sh
adduser 501 tim
bash$
```


© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

Not just ASCII

While it will get most people far, there's more to passing data between programs than just looking at the ASCII and pulling out columns from certain rows and mixing them around a bit. Consider the differences between:

```
find . | cpio -pdv /somewhere/else
```

and:

```
tar cvpf - . | tar xpf - -C /somewhere/else
```

In the former case, all that's being passed to cpio is a list of files, such as './file1 ./file2' and so on. In the second case, the data flowing through the pipe is a tar archive – the contents of the files, whatever that may be, in a binary archive format.

SHELL FEATURES

Backticks

Instead of passing output into input like a pipe, it's possible to use one command's output as arguments to another. For example:

```
bash$ ls
CVS        nolines    shell.html   shell.latte.~1.5.~  shell.pdf
shell.txt
Makefile   shell.aux  shell.html~  shell.latte~        shell.ps
Makefile~  shell.dvi  shell.latte  shell.log           shell.tex
bash$ ls | wc -w
     16
bash$ nofiles='ls | wc -w'
bash$ echo $nofiles
16
bash$
```

Here we have the 'ls | wc -w' being executed first, then the results ('16') put on the command line so it becomes a simple 'nofiles=16' command.

There are two problems with this:

1 You can't nest backticks: if you were to want to do something like:

```
ls 'dirname 'which ssh''
```

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

then it just won’t work. For this purpose, we have the $() construct instead:

```
ls $(dirname $(which ssh))
```

2 Most shells have restrictions on the maximum number of arguments you can have to any one command. For example:

```
cd /var/spool/mqueue
        rm *
```

is quite likely to fail if you have a substantial mail-queue. The answer to this is **xargs**. As a general rule, a command of the form

```
command 'command2'
```

is equivalent to:

```
command2 | xargs command
```

The way xargs works is to take each line in the input stream and substitute the placeholder (by default, end of line) with each of those lines in turn.

Adding users in batch, revisited

The above command (**find * -printf**) has a fatal flaw: if the directory in which you run it has lots of directory entries, the shell is going to expand the ‘*’ into a list of matching directory entries, and then complain it’s too long (typically 4096 words, tops). The following is an improved version:

```
bash$ find . -type f -maxdepth 1 -printf "adduser %u %p; echo %p |
passwd —stdin %p\n" | sed -e 's/.\///g'
```

(and |sh to execute it for real). The shell does not have to expand any global characters (* and ?), so will not complain. However, all the lines that ‘find .’ returns will start ‘./’, so this pattern is removed with sed.

For-loops

The shell has the ability to repeat commands; in csh, the appropriate syntax is:

```
csh> foreach i (*)
foreach? echo $i
```

 © 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

```
foreach? end
CVS
Makefile
Makefile~
shell.aux
shell.dvi
shell.html
[snip]
shell.txt
csh>
```

In all the other shells, the syntax is:

```
bash$ for i in *
> do
> echo $i
> done
CVS
Makefile
Makefile~
shell.aux
shell.dvi
shell.html
[snip]
shell.txt
bash$
```

As an additional bonus, zsh supports a nice syntax:

```
zsh% foreach i (*) {
> echo $i
}
```

All this does is the same as an 'ls -1'. However, this is merely the tip of the iceberg, when you can use it to run a filter on a load of files. For example, starting from this directory:

```
bash$ pwd
~/Pictures/Skye/3
bash$ ls
cnv00003.tiff  cnv00010.tiff  cnv00017.tiff  cnv00020.tiff
cnv00031.tiff
cnv00007.tiff  cnv00015.tiff  horse.tiff     cnv00025.tiff
cnv00032.tiff
cnv00008.tiff  cnv00016.tiff  cnv00019.tiff  scenery.tiff
cnv00033.tiff
```

It is easy to normalize the naming scheme down to a numeric system, and, for that matter, change the file format in bulk, too:

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

```
bash$ c=0; for i in *
> do
> convert $i newname$((c++)).jpg
> done
```

The resultant set of commands run is:

```
convert cnv00003.tiff newname0.jpg
convert cnv00010.tiff newname1.jpg
convert cnv00017.tiff newname2.jpg
convert cnv00020.tiff newname3.jpg
convert cnv00031.tiff newname4.jpg
```

and so on. This way it is trivial to build a script that takes a directory load of images (eg from a digital camera), and creates a set of thumbnails, where each thumbnail is defined as being 48*32 in size, named 'TN_something.jpg'; an index.html can then be produced in batch mode as well, or even better, an index.php3 that generates HTML on-the-fly, searching for TN_*.jpg and building a table of pictures, each of which is a link to the bigger picture. No 'magic thumbnail generator' software is required, just use your AIX/unix/ linux shell to fuller advantage.

MORE EXAMPLES

Xargs in action

To remove lots (>4000) of files in a directory, you can try:

```
rm /some/directory/*
```

but the chances are it will fail with an error about too many arguments.

Instead, try:

```
ls -1 /some/directory | xargs -n 200 rm
```

which will batch-up the incoming lines into groups of 200 and feed them all to rm.

An example of it not all working

This command:

© 2002. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

```
sudo echo 127.0.0.1 localhost > /etc/hosts
```

is probably not what someone intended to do; the file /etc/hosts is only writable by root, at least normally, which explains the call to *sudo* at the start of the line; however, what happens is that the shell parses this, splitting first on the '>' sign, so that the extent of the 'sudo' command is the end of 'localhost', not including the >. If you want to do the above correctly, you'll be best off with the following instead:

```
su root -c 'echo 127.0.0.1 localhost > /etc/hosts'
```

Exploiting shell parsing

Consider the commands:

```
bash$ ssh otherbox "find . | cpio -H tar -ov" | gzip -c >> /dev/tape

bash$ ssh otherbox "find ."| cpio -H tar -ov  | gzip -c > ~/backups/
foo.tgz
```

In the first case, the shell picks up everything between the "", as one 'word' argument to the **ssh** command, along with the parameter 'otherbox'. The results are a tar archive that comes through the **ssh** command, with the resulting tarball being gzipped and appended to tape. In other words, it will back up your remote homedirectory down to local tape.

In the second case, however, only the 'find .' bit is executed remotely; the **cpio** and **gzip** are both running locally here. The result is that only those files listed on the remote host and present locally are backed up.

Surprisingly enough, there is a valid use for this construct: if you're subsequently going to synchronize files from the remote host and want to know what's going to be overwritten locally, maybe taking a back-up first, then getting the list of files from the remote box may make sense.

The way to understand these commands is to consider both where the command is being split by the shell, and what the nature of the data is between commands.

Tim Haynes
Open Source and Free Software Consultant (UK) © Xephon 2002

© 2002. Reproduction prohibited. Please inform Xephon of any infringement.

AIX news

IBM has begun shipping its Regatta eServer p690, which boasts a system-on-a-chip design, with each chip containing two 1GHz-plus processors, a large memory cache, and I/O interconnected with other components using a new distributed switch design.

Using tools derived from IBM's Project eLiza programme, it includes multiple layers of self-healing capabilities designed to allow the system to continue operating through component failures and system errors.

It can be operated as a single large server or divided into as many as 16 virtual servers, running any combination of AIX 5L and Linux.

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

* * *

IBM has announced Tivoli Web Services Manager V1.7, a Web performance management tool, now including enterprise application performance management, which comes via a new level of Tivoli Application Performance Management and addresses the performance management of both Web-based applications and enterprise applications. It runs on AIX, Solaris, and Windows NT/2000.

Many enhancements are focused on addressing a wider range of Internet and enterprise transactions, ease of use, and industry standards. Also announced is support for nine additional languages by Tivoli Web Services Analyzer. The same languages are also provided in Tivoli Web Services Manager.

With Version 1.7, the ability of the Synthetic Transaction Investigator to work with client-side-generated dynamic content is improved. In WebSphere environments where WebSphere Site Analyzer is deployed, the software can now provide reporting that correlates the performance metrics it collects with the Web metrics from WebSphere Site Analyzer, showing how Web site traffic is affecting Web site performance.

There are also more granular administrative roles so that, in sites where different people will be responsible for different components, it's now possible to define administrators with unique administrative responsibilities.

For further information contact your local IBM representative.
URL: http://www.tivoli.com/products.

* * *

IBM Global Services will add Veritas products to its data infrastructure services for major platforms including AIX, Linux, and Windows, providing consulting, implementation, support, and maintenance services as part of its multi-vendor coordinated point of contact for IT services tailored for joint customers.

The full Veritas product range will be available, providing management for disaster recovery, data protection, storage virtualization, high availability, and storage-area networking.

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

* * *