ZIFF-DAVIS
THE COBB GROUP

November 1997 • $11.50
Vol. 3 No. 11

INSIDE SOLARIS™
Tips & techniques for users of SunSoft Solaris

in this issue

Visit our Web site at
http://www.cobb.com/sun/

Recording terminal sessions

by Alvin J. Alexander

Over the years, I've found that every good system administrator with whom I've worked keeps good notes when solving complex or unusual problems. Keeping notes helps in Solaris, since it can be a complicated operating system. When a problem occurs infrequently, you may find it difficult to remember which solution fixed the problem the last time.

If you keep good notes, you needn't reinvent the wheel every time a problem recurs. Maintaining good records also makes training an assistant that much easier, because your assistant can see exactly what you did to solve a particular problem. Keeping good records can mean more free time to work on other problems.

Simplifying note taking

However, when you're solving a problem, you're concentrating on finding the solution—not documenting your path. In addition, solving the problem, especially a complicated one, often takes more than one attempt. Once you've solved the problem, you may find it difficult to remember the exact sequence of steps you used. You're often forced to jot down notes on a scrap of paper while you're working on the problem

Also, taking notes while you're solving a problem can divert your attention from the problem itself, causing you to make mistakes. It would be much better if Solaris could take your notes for you. Luckily, Solaris provides a great tool to help you out—the `script` command. Using the `script` command, you can record the *exact* commands you entered to fix a problem. Not only does this command record everything you type, it also records every response that Solaris makes. Once you use the `script` command to create a detailed record of your actions, you can annotate it and review it at any time in the future to see the exact steps that were necessary to solve a problem.

Using the script command

The `script` command is easy to use. Here's a typical scenario: You're working at your desk, and a problem pops up. You wish you could get someone to record the steps to this solution. Then, an assistant can take care of it if it recurs. But you just don't have the time to write it down. Fear not; here's where the `script` command comes in to do the record-keeping for you.

For example, suppose you need to use the Solaris `format` command and you want to take notes. Before issuing the `format` command, use the `script` command to record all of your terminal input/output. In our example, we'll name our log file */tmp/format_session*:

SOFTBANK
ZIFF-DAVIS

INSIDE SOLARIS
Tips & techniques for users of SunSoft Solaris

Inside Solaris (ISSN 1081-3314) is published monthly by The Cobb Group.

Prices
U.S. ..... $115/yr ($11.50 each)
Outside U.S. ..... $135/yr ($16.95 each)

Phone and Fax
US toll free ..... (800) 223-8720
UK toll free ..... (0800) 961897
Local ..... (502) 493-3300
Customer Relations fax ..... (502) 491-8050
Editorial Department fax ..... (502) 491-4200
Editor-in-Chief ..... (502) 493-3204

Address
Send your tips, special requests, and other correspondence to:

The Editor, *Inside Solaris*
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220
Internet: inside_solaris@zd.com.

For subscriptions, fulfillment questions, and requests for group subscriptions, address your letters to:

Customer Relations
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220
Internet: cobb_customer_relations@zd.com

Staff
Editor-in-Chief ..... Marco C. Mason
Contributing Editors ..... Al Alexander
Production Artists ..... Margueriete Winburn
Natalie Strange
Editors ..... Karen S. Shields
Joan McKim
Publications Coordinator ..... Linda Recktenwald
Product Group Manager ..... Michael Stephens
Circulation Manager ..... Mike Schroeder
Managing Author ..... Eddie Tolle
Publisher ..... Mark Crane
President ..... John A. Jenkins

Back Issues
To order back issues, call Customer Relations at (800) 223-8720. Back issues cost $11.50 each, $16.95 outside the US. We accept MasterCard, Visa, or American Express, or we can bill you.

Postmaster
Periodicals postage paid in Louisville, KY and additional mailing offices.
Postmaster: Send address changes to:

Inside Solaris
P.O. Box 35160
Louisville, KY 40232


```
root:/home/al> script /tmp/
format_session
Script started, file is /tmp/
format_session
#
```

The `script` command tells you that it started and confirms the name of the log file it will use. Please note that if you don't specify a filename, the `script` command will create a file named *typescript* in your current working directory.

Also, after you enter the `script` command, your shell may change. When `script` begins, it starts a new shell of the type specified by your `SHELL` environment variable. So if you're in a different shell, be aware that your environment can change. In this example, we're using the root account to run the `format` command, since our day-to-day account has insufficient permissions to access the raw disk devices. Because we like the Korn shell, we usually run `ksh` after logging into the root account. So when we run `script`, we're already using the Korn shell with the prompt `root:/home/root>`; after running `script`, the prompt changes to the Bourne shell's familiar pound sign (#).

After issuing the `script` command, we can now focus on solving the problem at hand. To illustrate the operation of the `script` command, we'll start the `format` command and enter zero (0) to choose the first disk. (Please note that if we don't have superuser privileges, we won't have the option of selecting a disk.) Then, we'll type *quit* at the main menu to return to the shell prompt. Remember that at this point, we're still in the environment of the `script` command. Now that we've finished, we exit the `script` subshell and return to the previous command-line shell by typing `exit` or pressing [Ctrl]D:

```
# exit
Script is done, file is /tmp/
format_session

root:/home/al>
```

As the subshell exits, the `script` command reminds us of the name of the file we just created and ends, returning us to our normal shell command-line prompt.

Making your notes useful

Now that you have a copy of your terminal session, you can clean it up and insert any notes describing special cases or the reasons that you made certain decisions. To do this, you can edit the */tmp/format_session* file.

But be forewarned: Because the `script` command records everything you type and everything that's displayed onscreen, you'll see things in the script output file you may not have seen before, such as backspaces, carriage returns, and other control characters. For example, take a look at our sample `format` session shown in Figure A.

See that section in blue? That's where we accidentally mistyped the `format` command, then corrected it—each `^H` is a backspace character. You'll also notice that each line ends with a `^M` character: These are the carriage returns that you send to the computer whenever you press the [Enter] key and that Solaris sends back to you every time it prints to the screen. Normally you don't see these characters—they're hidden from you—but they do exist. Because they're sent to your terminal, the `script` command also sends them to your session file.

Actually, in a typical script session, the backspace and carriage return characters are the least of your worries. If you use any of the screen-oriented utilities, such as the `vi` editor, you'll really see some strange characters in your output file because these screen-oriented commands send a lot of escape sequences to your computer screen to control your display. The `script` command saves each of these characters into your session file, which can make editing your terminal session a bit tricky. For some tips on cleaning up the text file before you

(continued on page 4)

Figure A

```
Script started on Tue Aug 19 08:14:42 1997
# fomra^H ^H^H ^H^H ^Hrmat^M
Searching for disks...done^M
^M
c0t2d0: configured with capacity of 2.71GB^M
^M
^M
AVAILABLE DISK SELECTIONS:^M
       0. c0t1d0 <DEFAULT cyl 2045 alt 2 hd 21 sec 99>^M
          /pci@0,0/pci9004,7178@b/cmdk@1,0^M
       1. c0t2d0 <SEAGATE-ST43400N-1028 cyl 2735 alt 1 hd
21 sec 99>^M
          /pci@0,0/pci9004,7178@b/cmdk@2,0^M
Specify disk (enter its number): 0^M
selecting c0t1d0^M
[disk formatted]^M
Warning: Current Disk has mounted partitions.^M
^M
^M
FORMAT MENU:^M
        disk       - select a disk^M
        type       - select (define) a disk type^M
        partition  - select (define) a partition table^M
        current    - describe the current disk^M
        format     - format and analyze the disk^M
        fdisk      - run the fdisk program^M
        repair     - repair a defective sector^M
        label      - write label to the disk^M
        analyze    - surface analysis^M
        defect     - defect list management^M
        backup     - search for backup labels^M
        verify     - read and display labels^M
        save       - save new disk/partition definitions^M
        inquiry    - show vendor, product and revision^M
        volname    - set 8-character volume name^M
        !<cmd>     - execute <cmd>, then return^M
        quit^M
format> quit^M
# exit^M

script done on Tue Aug 19 08:15:35 1997
~
~
"/tmp/format_session" 40 lines, 1367 characters "typescript"
12 lines, 462 characters
```

Now we can use an editor, such as vi, to edit our captured terminal session.

Removing escape characters from your text files

So you have an ASCII file that you want to edit. But what if it contains so many control characters that you find difficult to read what's going on? Figure A shows a section of a `vi` session as captured by the `script` command.

Before you begin editing the file, you might want to clean it up. Solaris provides a tool, the `col` command, that will remove many of the extraneous characters from a file. You simply pipe the resulting file through the `col` command. This command is intended as a print filter to strip out basic formatting codes from a stream, so you can print the stream on a printer that doesn't accept all the formatting codes.

The `col` command doesn't understand all the possible escape sequences, so you'll see some unwanted characters leaking through. Use the `-b` switch to remove all backspace characters and partially repair the line. Please note that if you backspace over characters and replace them with white space, the old characters show through. For example, we'll type 10 underscores here, then 10 backspaces. On the screen, we've erased the line. As far as the `col` command is conceived, though, we're at the start of 10 underscores.

```
# script test
Script started, file is test
# __________^H^H^H^H^H^H^H^H^H^H a b c d e f g h
sh: a: not found
# exit
Script done, file is test
# col -b <test
Script started on Tue Aug 19 08:39:22 1997
# a_b_c_d_e_f g h
sh: a:  not found
# exit
script done on Tue Aug 19 08:39:41 1997
```

Now, when we type the letters *a* through *h* separated by spaces, the `col` command replaces the previous characters (underscores, in this case) with the letters you type. However, if you type white spaces, such as a space or tab, the original characters remain. While the `col` command is a useful tool for cleaning up extraneous characters in your ASCII files, it's definitely not perfect.

Figure A

```
-rw-r--r--   1 root     other^[[10C0 Aug 19 09:04 test^M
-rw-r--r--   1 marco    source     48825 Aug 16 00:33 trsf8001.zip^M
-rw-r--r--   1 root     other^[[8C277 Aug 19 08:34 typescript^M
~
^H~
^H~^[[H^[[49B"test" 20 lines, 1227 characters^[[H
^[[M^[[44B~^[[K^[[5;1H^[[M^[[44B~^[[5;1H^[[M^[[44B~^[[5;1H-rw-r^[[P^[[P^[[P-^[[P
 ^[[P ^[[P^H^H^H^H^[[A^[[Arwxr-x
^M^[[45B:wq^M"test" 17 lines, 1030 characters ^M^M
```

Sometimes an ASCII file can be almost unreadable because of the escape characters found in the file.

begin editing and annotating it, read the sidebar "Removing Escape Characters from Your Text Files" on page 3.

Whether or not you strip the file of its extended ASCII characters depends on the problem you're trying to solve. If you're debugging an application that's trying to manipulate the screen, you may need to keep these extended ASCII characters so you can see them. However, in our example case, where we're simply trying to record the dialog with Solaris' `format` command, we'd strip all the extended ASCII characters out of the file before viewing it.

After the rough-cleaning, you can trim the log file to remove any dead-ends you might have encountered while trying to solve the problem and describe what you were trying to do in some annotations. Be sure to document any assumptions you made about your environment while working, since these may change in the future, and you may need to change things the next time you solve the problem.

Other uses for the script command

Other than simply documenting your tracks, `script` command offers some additional uses as well. For example, if you want to write a shell script to perform a particular job, you can often start by doing the job manually and edit your terminal session to make a rudimentary shell script. Then, add error checking and other features as you require them.

To illustrate further, I once used the `script` command for a different purpose: As the administrator of a large network, I had a security concern with a network user. I discussed the problem with management, and after diplomatic discussions with the user failed, management and I decided to insert the `script` command, with a few modifications, into that user's startup files. We recorded the user's sessions for a few days until the problem was resolved.

Conclusion

As you can see from this article, you can use the Solaris `script` command as a terrific session-recording tool to keep better notes for solving problems. Keeping great notes will help you to solve recurring problems more easily and will also help you train assistants, who can see the exact steps you've followed to solve many of your system-administration problems. ❖

SYSTEM-ADMINISTRATION TOOL

What's the top priority on your system?

When your system is running slowly, you must find out why so you can fix the problem. Your first step might be to use the `ps` command to discover which processes are actually running on the system and to follow it up with the `sar` command. With some practice, you can use `sar` to recognize when a problem is getting ready to occur and identify it. However, `sar`'s output can be a bit difficult to interpret. In addition, `sar` shows only a summary of your system's activity; it doesn't show you which process is using the resources.

Using the top command

Fortunately, a third-party utility, named `top`, can help you get more information about your system. The `top` utility shows you which processes on your system are consuming the most CPU time and how much RAM they're taking. (In the case of a tie in CPU usage, the utility sorts the processes by RAM usage.) The output of the `top` command is arranged in an easy-to-read format, as shown in Figure A.

The `top` command, when run on an intelligent terminal, will run continuously, showing you a summary of the system's performance at the upper part of the screen and a sorted list of the top CPU consumers at the bottom. As the system operates, you'll see processes moving up and down, and appearing and disappearing on this display. The screen update interval defaults to five-second increments, though you can override the default to use any value you like.

Here, you can see that the process consuming the most RAM is the `gzip` command run by the `root` account. Also, the bottom four

CPU users on our list are consuming no CPU time. Since the bottom processes use the same amount of CPU time (i.e., 0), they're sorted by RAM usage.

The screen is updated every five seconds (by default) to show you a new picture of what has happened over the last five seconds. If you want to select a different number of processes, just specify the number on the command line. If you'd like a different update rate, you can specify the new rate with the `-s` switch. So, if you want to watch the top five processes at one-minute intervals, you can run the command

```
# top -s 60 5
```

Caveats

Please note that any system-monitoring tool affects the system you're running it on, so you don't want to run `top` all the time on a heavily loaded system. It'll just slow everything down. Figure A shows that the `top` program is consuming six percent of the CPU while it's running. On a lightly loaded system, that won't matter so much, but you shouldn't run programs if you're not going to use the information they produce. Run `top` only when you need it.

Time to use top

So when *do* you want to run `top`? Since `top` displays information about the CPU and RAM requirements, you'll probably want to run `top` whenever you suspect there's a CPU or RAM shortage and you want to find the culprit. If you notice that the system's performance is lower than normal, you might want to use the `uptime` command to find out what the current system load is, like this:

```
# uptime
  9:54am  up 17 day(s), 8 min(s),  5 users,
load average: 0.34, 0.13, 0.07
```

The `uptime` command shows you the load average for the previous one, five, and 15 minutes. The *load average* is simply the average number of processes waiting to be run over the sampling period. In the last minute, during roughly one-third of the time when one process was running, another was waiting to run. Thus, the higher the number, the more heavily your system is loaded.

Different systems can handle different loads before the system gives an unacceptable response. A large factor in determining an acceptable load is user-perception: Does the system feel too slow? What one person may accept as merely adequate performance, another may be very pleased with. In general, if your system load is continually higher than two or three, your system probably needs some tuning.

If your system has a light load and plenty of idle time, `top` probably won't tell you anything you can use. If the system is performing poorly in this situation, then your system is most likely I/O bound, and you'll want to use `sar`, `vmstat`, or `iostat` to find the problem. (Of course, there are exceptions to every rule. If one of your processes has a huge memory demand and is randomly accessing parts of its memory, this situation can force the system to thrash, causing lots of I/O to the swap surface. If you suspect this is the case, `top` may be able to help locate the culprit—you'd look at the RAM column, rather than the CPU column.)

Figure A

```
last pid:  2722;  load averages:  1.10,  0.97,  0.79                    07:32:22
67 processes:  64 sleeping, 1 running, 1 zombie, 1 on cpu
CPU states:  0.0% idle, 89.6% user, 10.4% kernel,  0.0% iowait,  0.0% swap
Memory: 41M real, 1276K free, 35M swap, 61M free swap

  PID USERNAME PRI NICE  SIZE   RES STATE   TIME   WCPU    CPU COMMAND
 2722 root     -25    0 1000K  764K run     6:17 93.06% 92.81% gzip
 2653 root     -25    0 1216K 1008K cpu     1:21  6.09%  6.12% top
  554 root      31    0 1412K  776K sleep   0:04  0.02%  0.19% nmbd
  320 marco     24    0 4964K 1056K sleep   0:13  0.06%  0.09% dtterm
  254 root      34    0   12M 1136K sleep   1:46  0.07%  0.07% Xsun
  204 root      33    0  732K  412K sleep   0:00  0.01%  0.01% utmpd
  352 marco     23    0 6432K  956K sleep   1:34  0.00%  0.00% dtfile
  319 marco     34    0 5752K 1160K sleep   0:15  0.00%  0.00% dtwm
   88 root      34    0 1120K  388K sleep   0:15  0.00%  0.00% cryptorand
 1068 root      33    0 1044K  872K sleep   0:11  0.00%  0.00% bash
```

The `top` utility shows the processes consuming the most CPU time on your machine, in descending order.

What to do with the information

Once you run `top` and find out which jobs are consuming all the CPU, you need to decide what to do about it. For a system with a light load and plenty of idle time, there's really nothing you can do. In general, stopping or killing processes won't accomplish much for a lightly loaded system, unless you stop a process that's thrashing the system.

On the other hand, if you have a couple of large CPU-hungry applications contending for CPU, you might want to lower their priority so that other jobs can get more CPU time. Another option is to stop some of the CPU hogs, restarting them when some of the ones you left running finally complete. This step will help give your system a boost.

Every now and then, you'll find that someone started a job, forgot about it, and started it again. In this case, you can probably kill all but one of the jobs, but be sure to ask the user which one(s) he wants to keep (in case the jobs were started with different parameters). If your user has no preference, you should probably kill the jobs with the least accumulated CPU time, as they're probably further from being finished.

Where to get top

Now that we've whetted your appetite, let's see where you can get a copy of `top` for your system. First, do you want `top` in executable or source-code form? If you elect to get a precompiled copy, you must be satisfied with the options selected by the person who compiled it.

One of the best places to get precompiled versions is the Solaris 2.5 freeware site at *http://www.sunsite.univalle.edu.co/Solaris/Solaris_2.5_nof.html*. Here, you can find packaged versions that are ready for installation. Just download, install, and use. The site also has a convenient link to a page for Solaris x86 users. Another site you might want to investigate is *http://www.sunsite.unc.edu/pub/solaris/freeware*. This site contains a source-code archive, as well as compiled versions of various programs for x86, SPARC, UltraSPARC, and a new directory for Solaris 2.6 versions. (However, it doesn't yet include a version of `top` for Solaris 2.6.)

If you get the source code, you must compile it, which requires that you have a C compiler on your system. (If you don't have a C compiler, you may as well get a copy of `gcc` while you're on the Internet.) On the plus side, you have the opportunity to select the configuration options you want to use when you build your own copy.

Building top

If you've gotten `top` as source code from the Internet, you'll have to compile it. Luckily, `top` is pretty easy to compile and install. We downloaded the source code to `top` v3.4 and followed the instructions in the *INSTALL* file. Specifically, you must run the Configure script, which sets up the file *Makefile*. (This file tells the compiler how to build `top`.)

The Configure script will ask you some questions. For the most part, you can accept the default responses given in brackets. We'll walk through the Configure question-and-answer session, but we'll trim most of the output of Configure to keep it short. First:

```
What module is appropriate for this machine?
```

You should answer with *sunos54* if you're running Solaris 2.4 through 2.5.1, or *sunos5* if you're running an older version of Solaris. Configure asks if you're sure that this is the right module. Just press [Enter].

Next, it will ask you for the path to the Bourne shell and the name of your AWK interpreter. Go ahead and accept the defaults for these. Then, Configure asks for the name of your C compiler. If you're using Sun's C compiler, you can accept the default value of `cc`, though `gcc` users will want to specify `gcc`.

At this point, the Configure script will ask several other questions, to which you can accept the default:

```
Installer [./install]:
Compiler options [-O]:
LoadMax [5.0]:
```

Configure will also ask you how many processes you want to show on the screen by default. Normally, Configure will use 15, but you can select any value you like. Here, we're telling it to use 10:

```
Default TOPN [15]: 10
```

Configure then asks you how many processes to display when the output is a dumb terminal. We'll use 10 here, too:

```
Nominal TOPN [18]: 10
```

Next, we can choose the time interval between displays. The default is five seconds. If you choose a shorter time period, `top` will consume more of your CPU, while a longer time period will decrease `top`'s demands accordingly. (If you intend to run `top` continuously as a status display, even though we don't recommend it, you should use a relatively long time period, such as one or five minutes.)

```
Default Delay [5]:
```

Since we're running Solaris, the Configure script then remarks:

```
It looks like you have conventional passwd
file access.  Top can take advantage of a
random access passwd mechanism if such exists.
Do you want top to assume that accesses to the
file /etc/passwd are done with random access
rather than sequential? [no]:
```

Now comes the interesting part. Configure compiles a prime number generator so it can generate the size of an internal hash table. It does so to make access to the user name as fast as possible. In order to minimize the RAM requirements, the Configure script defaults to selecting the first prime number that's at least twice as large as the number of users in your password file.

You can accept this value, but if you're compiling it on a test machine and you intend to use it on a machine with a lot more users on it, you'd be better off choosing a prime number based on the size of the */etc/password* file on the machine that has the most users. Here, it suggests a hash-table size of 29, but since our destination machine has 60 users, we'll use 127 as our hash-table size:

```
Enter the hash table size [29]: 127
```

Next, Configure wants to know whether it may install the `top` program as an `SUID` program with `root` privileges. Generally, using `SUID` programs is a security risk, so be sure to install `top` in a location where only the `root` account has access to it and take the standard precautions you would with any `SUID` program. We just accept the defaults for the owner, group, and mode:

```
Owner [root]:
Group owner [bin]:
Mode [4711]:
```

For our installation, we'll be a little pickier when we specify the location to install the program:

```
Install the executable in this directory
[/usr/local/bin]: /usr/bin
```

You can now accept the default values for the rest of the Configure script:

```
Install the manual page in this directory [/usr/man/man1]:
Install the manual page with this extension [1]:
Install the manual page as 'man' or 'catman'
[man]:
```

When Configure is finished, it builds the appropriate `Makefile` and tells you how to create and install `top`:

```
To create the executable, type "make".
To install the executable, type "make install".
```

So, run `make`, then `make install`, and you'll have a copy of `top` installed on your system.

Installing a precompiled version

If you download a copy of `top` from the Internet, please be sure you trust the source. Since `top` is normally an `SUID` program on Solaris machines, it's a potential security hole. A malicious source could create a version that will erase files on your machine or grant surreptitious access to your machine.

Since `top` can be built and packaged in multiple ways, we can't give explicit instructions for installation here. Be sure you read the installation instructions and/or the script before you install `top` on your machine from another source.

Summary

System monitoring is one of the hardest system administration skills to learn, since there are so many interrelated factors involved. However, before you can learn to tune a system effectively, you need to find out just what the system is doing at any given time. The `top` utility is a great tool that you should get and use—it'll give you some insight as to what's happening in your computer. ❖

If you have any ideas for an article, any concerns to discuss, or any tips and and techniques you'd like to share, please e-mail us at *inside_solaris@zd.com*.

Changing the root account's shell

No matter what your favorite shell is, you should never change the root account's starting shell. If you do, your system probably won't boot properly. Why? When you start your system, it executes a series of Bourne-shell scripts that initialize all the subsystems. Since your system uses the root account to do so, if you change the root account's shell, the new shell may have a slightly or radically different interpretation of the script.

However, other shells are more powerful and easier to use than the Bourne shell, so few people still use it for anything other than writing shell scripts. But how do you use another shell for system administration? In this article, we'll show you a simple modification to your *.profile* file that will allow you to automatically start a new shell.

Manually start a new shell

Obviously, one way to use a new shell on the root account entails manually invoking the desired shell each time you log in. For example, to use the C shell, your login session would look like this:

```
UNIX(r) System V Release 4.0 (Pinky)

login: root
Last login: Tue Aug 19 07:21:43 on pts/2
Sun Microsystems Inc.   SunOS 5.5       Generic
November 1995
# csh
Pinky# _
```

The problem with this approach is that now you've got two processes in memory: the original Bourne shell you used to log into the system and the C shell. You can save RAM and dispense with a process by replacing your Bourne shell with the C shell. You do so by starting the C shell with the `exec` command. This way, when you've finished with your shell, you needn't log out twice:

```
Pinky# exit
# exit
```

An automated approach

What you really want, however, is for the account to automatically invoke the desired shell when you log in. Therefore, you require a method that will differentiate between when the root account is starting up the system and when it's running normally. You meet these needs by recognizing that Solaris performs the system startup tasks in run-levels 1 and 2, while interactive use is available at run-level 3. Thus, adding a bit of code in your *.profile* file to check the run-level and invoke the correct shell comes close to giving you what you want:

```
RUNLEVEL=`/usr/bin/who -r | awk '{ print $3 }'`
if [ 3 = $RUNLEVEL ]
then
    exec /bin/csh
fi
```

Here, we use the `who` command to report the current run-level, select the third column (the value we want) with `awk`, and assign the resulting value to the `RUNLEVEL` variable. Then we test `RUNLEVEL` to see whether it holds a 3. If so, we go ahead and `exec` the C shell.

If you want to use only the C shell, then that's all you'll need to successfully start the shell. However, for users of other shells, such as the Korn and Bash shells, this method doesn't quite work. Since both the Korn and Bash shells may also read the *.profile* file for startup commands, your account can freeze as each shell comes up and loads another copy of the shell infinitely.

In this case, you must know whether or not the shell is the login shell. To allow you to detect the login shell, when the `init` process starts your program, it prepends the shell's name with a hyphen and passes that name to your shell as its first argument. Thus, if your startup shell is the Bourne shell, the first argument would be `-sh`. In this case, you simply check for the hyphen at the beginning of your shell's name. This gives the final piece of the puzzle. You can use the code snippet shown in Listing A to start the shell you want when you log in.

Login customization

This technique has a minor shortcoming: Once you use the `exec` command to start a new shell, it's no longer recognized as your login shell. Normally, this quirk won't cause a problem, except when you perform a specific procedure at login time to customize your environment.

For example, if you use the C shell and normally use the *.login* file to customize your environment, you'll be disappointed that it won't use the *.login* file to read your customizations. (This isn't, after all, a login shell.) Of course, you could work around this by explicitly sourcing your *.login* file like this:

```
Devo% source .login
```

However, since you're trying to automate the process, a better workaround is to give an explicit parameter to the C shell when you `exec` it. Then, your `.cshrc` file can check whether it should source the `.login` file. Thus, you can change the `exec` statement to read

```
exec csh -s login
```

Please note that you need the `-s` parameter to tell the C shell to read its commands out of the standard input stream. Otherwise, the C shell attempts to open and read a file named *login*.

Now, since each C shell invocation reads the *.cshrc* file, you can explicitly source the *.login* file when you detect a new login attempt, like this:

```
echo "evaluating .cshrc"

# Is this a fake login shell?
if ( $#argv == 1 && $argv[1] == "login" ) then
   echo "sourcing .login"
   source .login
endif
```

When you log in, you'll start the C shell *and* get your customizations as well:

```
login:root
Last login: Thu Aug 21 09:05:33 on console
Sun Microsystems Inc.   SunOS 5.6       Beta
Update     August 1997
evaluating .cshrc
sourcing .login
Devo#
```

Why not just put all your customization information in *.cshrc*? You certainly can do so, but if you make extensive changes to your environment, your *.cshrc* file will get larger and larger. Since the C shell will read and parse this file every time you start a new C shell, it will take more time to start your shell. Since no one likes to wait any longer than necessary, you can put many of your customizations in the *.login* file, where your .cshrc file will be read only once—at login time—and not each time you start a new C shell, (whether from a script, command line, or whatever) so you'll save a little time.

Listing A: *A fragment of .profile*

```
# Is this a login shell?
case $0 in
    # Yes, it starts with a hyphen...
    -*)
        # Are we at the right run-level?
        RUNLEVEL='/usr/bin/who -r |   \
                  awk '{ print $3 }'`
        if [ 3 = $RUNLEVEL]
        then
            # Yes, start the desired shell
            exec /bin/csh
        fi
        ;;
esac
```

The Korn shell reads only the *.profile* file for customization information, so go ahead and put your information in the *.profile* file. Some people like to speed the startup of the Korn shell and often take advantage of the fact that the first argument to the login shell begins with a hyphen (as we did earlier to decide whether we could start the new shell). However, this technique won't be available to you, since the Korn shell won't be your login shell when you start it in this fashion. Instead, you can use the same technique we used for the C shell. To give it a parameter, just `exec` your Korn shell like this:

```
exec ksh -s login
```

Now, in your *.profile* file, you can check for the parameter to decide whether you're logging in:

```
# Is this a fake login shell?
if [ $# = 1 -a "$1" = "login" ];
then
    # Your Login-only stuff goes here
    fortune
fi
```

Other uses for this technique

There's another great use for this technique: If you can't get your system administrator to modify your account to use another shell, you can use this technique to select the appropriate shell when you log in. For example, the Bash shell, zsh, and other shells are available over the Internet, but your system administrator may not use them. Nonetheless, that doesn't have to prevent you from doing so! ❖

Sourcing files to change your environment

Have you ever wanted to write a program or script file that would change your current environment variables? For example, Listing A contains a simple script, named *AddLocal*, that attempts to modify the PATH variable to allow access to some programs in the */usr/local/bin* directory.

Listing A: *AddLocal shell script*

```
#!/bin/sh

# Allow use of GNU & other utilities
PATH=$PATH:/usr/local/bin
export PATH
echo "New path is: " $PATH
```

It would be great if you could use a script file to change your environment variables. But, you can see that it just doesn't work:

```
$ echo $PATH
/usr/bin:/etc:.
$ AddLocal
New path is:  /usr/bin:/etc:.:/usr/local/bin
$ gcc --version
ksh: gcc:  not found
$ echo $PATH
/usr/bin:/etc:.
```

How *can* you do it?

What went wrong? Why can't we change our environment variables in the script? Each time you execute a program or script from the shell, it starts a new process with its own copy of the environment—so your programs never actually have access to your environment variables, only to a copy of them. When we ran the *AddLocal* script, it changed its copy of the PATH variable and printed its new value. But when the script ended, it dropped us back to our shell with the previous value for PATH.

Don't worry: We do have a trick up our sleeve. (We don't bother telling you that something can't be done unless we have a way around it, do we?) It turns out that each of the standard shells that come with Solaris have a method of executing a list of commands in the current environment. For both the Korn and Bourne shells, you use a period (.) to tell the shell to read the named file and execute the list of commands in the current shell environment. If we want to use our *AddLocal* script to change our PATH, we do so like this:

```
$ . AddLocal
New path is:  /usr/bin:/etc:.:/usr/local/bin
$ echo $PATH
/usr/bin:/etc:.:/usr/local/bin
$ gcc --version
2.6.3
```

The . command simply reads the file *AddLocal* and executes each command as if you had typed it at the command prompt. So your *AddLocal* file doesn't even need to be a script—it only must hold valid commands.

In the C shell

If you use the C shell, you can use an almost identical workaround. The difference is you use the `source` command instead of using a period (.). Likewise, the command to set the PATH in the C shell has a different syntax. So our *AddLocal.csh* script looks like this:

```
# Allow use of GNU & other utilities
setenv PATH "${PATH}:/usr/local/bin"
echo "New path is: " $PATH
```

Now, we can execute our new script in the C shell, like so:

```
Devo% source AddLocal.csh
New path is:  /usr/bin:/etc:.:/usr/local/bin
Devo% gcc --version
2.6.3
```

Possible uses

This technique comes in handy if you perform several roles in your job. For example, if you do both system administration and program development, you may want to write a pair of command lists to set up each environment in the appropriate fashion.

You can also use this technique when you're debugging your shell startup scripts (e.g., *.profile*, *.login*). You'll find it's tedious to change your startup scripts, log out, and log

in repeatedly to test them. It's also quite dangerous—if you make a serious mistake, you may no longer be able to log in to your account properly. As an alternative, give your startup scripts a different name and use the `period` (.) or `source` command to execute them. Then, test your environment. Once it all works properly, you can replace your shell's startup files, or use the `period` (.) or `source` command in your shell's startup file to invoke the appropriate commands.

Closing notes

It's actually a good thing that scripts and programs can't modify your local environment without explicitly using the `period` (.) or `source` command to do so. Think of the confusion and security holes that would otherwise result ! ❖

A brief introduction to the sed command

It's a fact that different people have different favorite tools. When you're confronted with a problem to solve, you tend to think in terms of the tools you know. Therefore, the more tools you're comfortable with, the easier solving problems becomes.

The `sed` command is a very powerful tool with which you ought to familiarize yourself. Unfortunately, the `sed` utility intimidates people because it has a reputation for being difficult to learn and use. Actually, it isn't difficult to do either. Perhaps `sed`'s biggest fault is that a beginner may find reading and understanding a `sed` expression rather difficult. That's probably how it gained its reputation.

In this article, we'll introduce you to some `sed` basics, so you can begin exploring its many features. To that end, we'll show you a couple of the basic `sed` commands and turn you loose with them.

What does sed do?

First, we should probably describe the basic purpose of `sed`, so you can see where it fits in the world of UNIX tools. `sed` gets its name from its intended purpose: It's a *s*tream *ed*itor. Typically, you use `sed` to process a stream of data to either transform the stream or mine it for data.

When you start `sed`, you give it a script of commands to execute. Then, for each line in the input stream, `sed` reads a line into the working buffer, applies (in turn) the appropriate commands, and prints the resulting line to the standard output stream.

Some sed commands

Although `sed` offers you many commands to manipulate text, we'll just introduce you to a few of them here. For a complete list, please review `sed`'s `man` page. The command that you'll encounter the most is the `s` command, which performs text substitution. If you use the `vi` editor, then you're already familiar with the operation of this command.

The basic syntax for the command is

```
s/old/new/flags
```

where `old` represents the text pattern you're searching for, `new` represents the text you want to replace it with, and `flags` specifies the options you want for the search command. The `/` character separates the components of the command. Please note that you can use any character instead of the `/`, other than a backslash or newline.

For the `old` parameter, you can use any standard regular expression. So if you want to find a line that ends with a capital M, you can use the regular expression `M$`. The `new` parameter is simply the text you're substituting for the `old` parameter. The `flags` parameter is most often a `g`, when it's not omitted altogether. A `g` tells the `s` command to repeat the operation as many times as possible on the working buffer. If you omit the `flags` parameter, `sed` will perform the replacement only once. Thus, if you want to replace every instance of the word `flashlight` in a stream with the word `torch`, you'd use the command

```
s/flashlight/torch/g
```

The power of this command is limited only by your knowledge of regular expressions. (If you're rusty on your regular expressions, you may want to read the companion article "Using Regular Expressions in Solaris.")

Another often-used command is the y command, which allows you to replace one set of characters with another. The syntax for the y command is

```
y/oldchars/newchars/
```

The oldchars string must have no repeated characters and must have the same length as newchars. When sed runs the y command, it examines the working buffer, and each time it finds a character in the oldchars parameter, sed substitutes the corresponding character in the newchars parameter. As an example, if you have a document that's in mixed case, and you want only uppercase, then you can use the following command to map all the lowercase characters to uppercase:

```
y/abcdefghijklmnopqrstuvwxyz/
➥ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Quick Tip: If the only transformation you want to make is to replace one set of characters with another, you'll probably want to use the tr command instead of sed. The tr command not only translates character sets like sed, it also allows you to use shorthand notation, while offering some extras as well.

The last two commands we'll take a look at are the p and d commands. The p command simply tells sed to print the contents of the working buffer to the standard output. The d command tells sed to delete the working buffer, read in another line, and start processing at the first command in the list, ignoring any commands that may not have yet been processed. For example, if you have the list of commands

```
p
s/abc/xyz/
d
s/xyz/pdq/
```

then sed will print the working buffer immediately after reading it. It will replace the first instance of abc with xyz, if it exists, then delete the working buffer, loading it with the next line, and begin again with the p command. sed will never execute the second replace command.

Command addresses

If that's as far as sed went, then sed would be indisputably useful. However, sed has a feature that makes it far more powerful: You can specify addresses for the commands, which turn them on and off for parts of the stream. So you needn't perform every operation on every line of input.

There are three types of command addresses that sed uses: You can use a decimal number to specify a line number, a $ to specify the last line in a file, or a regular expression to locate a line that has a specific pattern in it. A regular expression used in this way is surrounded by / characters.

Most commands may have zero, one, or two addresses associated with them. (See the man page for details.) If a command has no addresses, then sed tries to execute the command for every line in the stream. If a command has only one address, then sed executes the command only for the specified line in the stream. sed treats two addresses as a range, first applying the command on the first line matching the first address and continually applying it to each line until the second address is satisfied, or until sed reaches the end of the file.

You can use command addresses simply by prefixing the command with the desired address(es). If you use two commands, you must use a comma or semicolon to separate them. A number specifies an absolute line number, and a regular expression must be enclosed by forward slashes.

Suppose for a moment that you want to replace every instance of the word the with XYZ beginning with the tenth line of the file and ending with the next line that contains the word onion. You could do so with the command

```
10,/onion/s/the/XYZ/g
```

Now you can see why sed scripts can be so difficult to read. If you're not familiar with command addressing, you'd be confused by the statement. Here, the 10 is the first part of the address, the comma separates the first address from the next, and /onion/ is the second address of the command, which specifies a regular expression that matches the pattern "onion." The actual command starts with the letter s, in the middle of the expression.

Now, what happens if you want to execute a particular set of commands on the same range? Must you specify the same range for each command? Thankfully, the answer is no. sed allows you to group commands together as a unit within braces ({ }). This feature allows you to specify your address range once, then enclose your command list in braces. Thus, if you wanted to expand the previous example to also change THE to xyz, you could do it with the following statement:

```
10,/onion/{s/the/XYZ/g;s/THE/xyz/g}
```

Using sed

Now that we have the basics out of the way, let's go into more detail on how to use the `sed` command, then try a couple of examples. First, let's look at how you can specify commands.

The easiest way to tell `sed` which commands you want to use is with the `-e` clause. After the `-e`, just type in the command you want `sed` to execute. You can specify multiple `-e` clauses if you want, or you can combine the commands into a single clause by separating the commands with a newline or semicolon.

Thus, these three commands do the same thing: They convert all curly braces to parentheses and delete any line that contains the word `zero`:

```
# sed -e 'y/{}/()/' -e '/zero/d'
# sed -e 'y/{}/()/;/zero/d'
# sed -e 'y/{}/()/
>         /zero/d'
```

When you use `sed`, you can either specify a list of files to operate on, or you can use it as a filter. In both cases, the output of the `sed` command is the standard output stream.

Quick Tip: If you use multiple input files, then you must be aware that when you use numeric command addresses, these addresses refer to the cumulative count of lines. Don't expect `sed` to reset the line count back to zero when it opens each file; you'll be disappointed. So be careful when using numeric command addresses.

If you're going to use just one `-e` clause, you needn't specify the `-e` part at all. If `sed` sees only a single command-line argument, it assumes that the argument is a command list and that you're using it as a filter. So, you could write the second command line above like so:

```
# sed 'y/{}/()/;/zero/d'
```

Quick Tip: You don't need to surround your command expressions with quotes, but it's a good habit to get into. The quotes prevent the shell from interpreting any special characters in the commands, then modifying the expressions before they're passed to `sed`. For example, if you don't quote the previous command, the Korn shell will complain like this:

```
$ sed y/{}/()/;/zero/d
ksh: syntax error: '(' unexpected
```

Other shells will complain differently. While you can learn the special symbols for each shell and quote the command expression when you use one, it's much simpler to always quote the command expression.

If you're going to use several `sed` commands to do a job, you may want to create a `sed` script. The simplest way to do so is to create a text file that contains your list of `sed` commands, then tell `sed` to use it with the `-f` switch. For example, suppose you find yourself making the same typos all the time, e.g., you use `teh` instead of `the`, `oen` instead of `one`, and `thier` instead of `their`. You could create a small text file named *myTypos* with the commands

```
s/Teh/The/g
s/teh/the/g
s/Oen/One/g
s/oen/one/g
s/Thier/Their/g
s/thier/their/g
```

Then, you can go ahead and type your document. When you want to fix your most common spelling *errerz*, you can pipe your new document through `sed` using your new `sed` script, like this:

```
$ sed -f myTypos <ToFix.doc >Fixed.doc
```

(Please note: This example needs more work to make it useful. As it stands, if it finds Teh as a part of word, it will 'correct' it; if you write about Tehran, you'll have to recorrect the document.)

Debugging sed commands

When you're working with `sed`, don't be shy about adding the `p` command periodically to print the current contents of the working buffer. This way, you can see what `sed`'s doing, and you needn't guess about what's happening.

One common mistake you should watch for involves command-address ranges, i.e., commands with two addresses. It's a common assumption that the second address specifies the line to stop processing the command(s) associated with the range, rather than the last line to be processed. For example, suppose you start each paragraph with four spaces. Whenever you find the word `shout` in a paragraph, you want to convert the rest of the paragraph to uppercase. To do this, you might think you could use the statement

```
/shout/,/^    /{ y/abcdefghijklm/ABCDEFGHIJKLM/
y/nopqrstuvwxyz/NOPQRSTUVWXYZ/ }
```

However, this statement will also capitalize the first line of the following paragraph.

Hopefully, you'll want to read the `man` page for the `sed` command and begin playing with it. Please be sure to read next month's article "Displaying a Directory Tree with `find` and `sed`" to see a practical example of how you might use `sed`. We again urge you to read the next article "Using Regular Expressions in Solaris," since `sed` relies so heavily on regular expressions.

Conclusion

The `sed` command is a tool that you're going to run into sooner or later. Either you're going to run into a problem that's ideally suited to `sed`, or someone is going to hand you a `sed` script that you'll want to modify for your own purposes. Admittedly, when you begin, you may find `sed` scripts hard to read. But once you become familiar with `sed`, you'll find it's really not that difficult to use. ❖

BEGINNERS SKILLS

Using regular expressions in Solaris

Of the many skills you'll use in Solaris, perhaps one of the most important is learning how to make the most of regular expressions. Many programs allow you to work with regular expressions: `vi` uses them to search and replace, `grep` uses them to find text in an input stream, `sed` uses them to decide which commands to execute, and even more programs let you use them to navigate through a file.

What's a regular expression?

So, if regular expressions are that important in Solaris, just what are they? A *regular expression* is a description of a piece of text. Many utility programs in Solaris use regular expressions to locate a particular part of a file or stream. Regular expressions can be very simple or very complex. The simplest regular expressions are literal strings: The string describes itself.

For example, say you're editing a file with `vi`, and you're looking for the word *tell*. You'd first type the / key, to instruct `vi` that you want to search for a regular expression, then you'd type *tell*, and press [Enter]. Then, `vi` searches for the next occurrence of the word *tell* through the file you're editing.

Sometimes, you don't know the exact character string you're looking for: If you capitalize it, for instance, you could miss the instance you were looking for. So you'd have to first search through your file for the word *tell*; if that search doesn't pan out, you would next search for *Tell*. It would be better if you'd search for either case with the same regular expression: You need a method to describe the strings in a more generalized way.

A bit of choice

You can add a bit of flexibility by describing some possible alternatives. Typically, you do this at the character level, though in some applications you may actually specify multiple regular expressions to search for. We'll concentrate on the character level for the purposes of this article.

If you have only a couple of selections from which to choose, you can specify the alternatives in a list and enclose the list in brackets ([ ]). Using our example again, we could search for the word *tell* with the expression

```
[Tt]ell
```

This regular expression matches either *T* or *t* followed by *ell*, so it'll find both *Tell* and *tell*.

You can put lists of characters to choose from in any location in your regular expression. So, if you want to find any instance of *master* or *mister*, you could use the expression

```
m[ai]ster
```

Ranges

If you're specifying a list of acceptable characters in brackets with a large range of possible characters, you needn't type each character if they're in a range. Instead you can use the lowest character in the range followed by a hyphen (-) followed by the largest character in the range, such as a-z. Even better, you can in-

termix ranges and normal characters. So, if you want to specify any legal hexadecimal digit, you could use the expression

```
[0-9a-fA-F]
```

Sometimes, it's simpler to specify all the illegal characters than the legal ones. You can do so if you use a caret (^) as the first character after the [in your list. This character tells the regular expression that the list specifies all the illegal characters. Thus, if you'll allow anything other than a digit or the letter M, you can simply specify:

```
[^0-9mM]
```

Repetition

When you want part of the regular expression repeated, you can specify this with the asterisk (*), which means that the previous character may be repeated zero or more times. For example, if you define a name as any capital letter followed by zero or more lowercase letters, you could use the expression

```
[A-Z][a--z]*
```

Or maybe you write C++ programs and like to start your program headers with a bar of hyphens using a double-slash comment. You might use the following regular expression to find the next function:

```
//--*
```

Please note that we use two hyphens followed by an asterisk. If we used only one, we'd find every double-slash comment in our program because the * says "zero or more" of the character. The first hyphen ensures that there must be at least one.

Special characters

When you're using a simple string as a regular expression, you just type in the string. Since we also need the ability to describe strings, Solaris assigns new meanings to some special characters. We've already seen most of these: The asterisk (*) allows you to specify that the last character may be repeated zero or more times, a period (.) will match any character except a newline, and the brackets ([ ]) allow you to specify one character of a list. Also, within the brackets, if the first character is a caret (^), you know that the list specifies the characters that may not be matched.

Two other special characters of note are the caret (when it's outside of the brackets) and the dollar sign ($). The caret at the start of a regular expression specifies the beginning of a line and allows you to locate strings based on the start of a line. The $ allows you tie a string to the end of a line when it's at the end of a regular expression. Otherwise, these symbols have no meaning, except the caret when it's used in ranges.

Sometimes, you just want to use a special character as a literal. In this case, you need to precede it with another special character, the backslash (\), which says that the next character isn't special, after all. So, if you want to put an asterisk in your regular expression, simply precede it with a backslash.

A brief example

Now, let's put some of this stuff together so you can see how it's all used. For our example, we'll use `grep`, and a file, named *Filelist*, containing a list of all the files on the system as our test file. We created this file in the root account, with the command

```
# find / -print >/Filelist
```

Suppose we know there's a file somewhere on the system that begins with a Q and ends with a d, and it's the name of a place. However, we don't remember anything more about the file. We can find it like this:

```
$ grep '/Q[^/]*d$' </Filelist
/usr/share/lib/zoneinfo/Australia/Queensland
/usr/dt/share/examples/dthelp/help/graphics/
QuickHelp.xwd
```

Aha! It was Queensland, and now we know where the file is. We began the expression with a /Q because we want to be sure that the file starts with a Q. Since a / precedes each filename, this construct only allows a match if Q is the first letter in the filename. Next in our filename, we can take any number of characters that don't include a / (which would start a new part of a filename). Finally, we use a d$ to say that we want a d at the end of our filename. Since we want to ensure that the d is at the end of the filename, we end the expression with $, telling it that no characters are allowed after the d.

Conclusion

Well, there you have it—a quick introduction to regular expressions in Solaris. These are

PERIODICALS MAIL

SunSoft Technical Support

(800) 786-7638

Please include account number from label with any correspondence.

only the basics, and some programs allow even more flexibility in specifying the regular expressions. You'll probably want to look over the `man` page for `regexp` for some of the basic rules. However, the `regexp man` page actually documents some functions used in Solaris, so you'll see a lot of information there that you may not be able to use. Keep in mind, the interesting parts are in the sections 'Basic Regular Expressions' and 'Characters With Special Meaning.' ❖

QUICK TIP

Using wild cards for file and directory name completion

In the article "Filename Expansion in the C and Korn Shells" in the September '97 issue, we showed you how to use command-line completion in the C and Korn shells. One reader pointed out that you can also use standard wild-card expansion for command-line completion. It's not as flexible, but it works in all the shells.

Here's the basic technique: You need to use only the part of the filename that's unique in the directory. So, if you have a long file or directory name that you want to work with, you need only a few characters. Suppose for a moment that you have a long filename that you want to work with, but you're not a great typist and don't want to type the whole name. You want to uncompress the file samba_1_9_16p8_tar.gz. Since you must specify only enough of the filename to be unique, you could use the command:

```
$ gunzip s*gz
```

if you have no other files that begin with an s and end with gz in your directory.

Suppose your directory contains the files:

```
$ ls
samba_1_9_15p8_tar.gz    samba_1_9_16p9_tar.gz
```

In this case, your command will expand to

```
gunzip samba_1_9_15p8_tar.gz
 samba_1_9_16p9_tar.gz
```

Oops—you're going to expand two files instead of only the one you want. In our previous article, we showed you some keystrokes that would show which files matched to the specified point. The wild-card technique is a similar trick: If you want to see if your proposed wild-card expression matches too many files, you can either type `ls`, as we did above, and inspect the output, or you can use the `echo` command, like this:

```
# echo /u*/s*
/usr/sadm /usr/sbin /usr/share /usr/snadm /usr/
spool /usr/src
```

Here, we were thinking that `/u*/s*` might be a good shorthand for the /usr/sbin directory. As you can see, we forgot about several other directories. Instead, we can use `/u*/sb*`.

Returning to our prior example, we want to expand one of two files in which the unique portion of the filename is in the center. Must we use the expression samba_1_9_16* to specify the file? No, since the shells allow us to put wild cards anywhere in a filename argument, we could use the following command to expand the latest version:

```
$ gunzip *16*
```

As you can see, it's much easier to type *16* than samba_1_9_16p9_tar.gz. So, the next time you want to take a shortcut and you're not in your favorite shell, remember that all you need to do is take advantage of wild cards to specify long file and directory names. ❖