

SUN INSIDE SOLARIS

Tips & techniques for users of SunSoft Solaris

in this issue

1
The secret of reading keyboard input one key at a time

4
Using tput to dress up your character-mode screens

7
Make a shell script mail you a summary

9
Don't clutter your system with core images!

10
Conditionally executing shell scripts

11
Hard and soft file links

14
Build a keyword index for man

15
Who's tying up the file system

16
Quickly switch between directories

Visit our Web page at
<http://www.cobb.com/sun/>

The secret of reading keyboard input one key at a time

By Al Alexander

Have you ever wished you could write a shell script that would wait for the user to press a single character and then proceed? Several years ago, we discovered a cool shell programming technique that lets you do just that. The premise is simple: You want to display a text menu to your end users, like the one shown in [Figure A](#). The user should only have to type one key to select the desired menu option, such as the letter C to start a CAD application.

Unfortunately, using standard shell programming techniques, it's not possi-

ble to type just one character to start an application—you must always type that character (such as the letter C), followed by the [Enter] key.

The desire to eliminate the requirement to press the [Enter] key led several people on a journey that required weeks of thought, research, and discussion before we finally discovered the proper technique. We'll discuss that technique—which requires only six lines of shell programming code—in this article. The technique presented here stretches one's knowledge of terminal I/O, the `dd` command, and I/O redirection and employs an

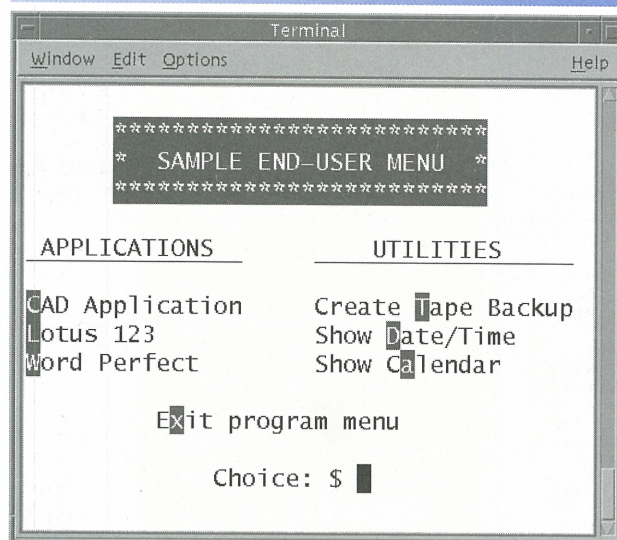
unusual method for invoking a shell script function.

Our technique is also very useful in other situations. For instance, if you've ever wanted to prompt an end user for a password and display asterisks (*) on the screen as that person types, then you'll find the starting point right here.

The solution

We discuss how to make a fancy menu, such as the one shown in [Figure A](#), in the

Figure A



This is a typical text-based menu for selecting which application program to run.

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@merlin.cobb.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: cr@merlin.cobb.zd.com.

Staff

Editor-in-Chief Marco C. Mason
Contributing Editor Al Alexander
Production Artist Margueriete E. Winburn
Editors Linda Recktenwald
Martha Bundy
Circulation Manager Mike Schroeder
Editorial Director Linda Baughman
VP/Publisher Lou Armstrong

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.

Advertising

For information about advertising in Cobb Group journals, contact Tracee Bell Troutt at (800) 223-8720, ext. 430.

Postmaster

Second class postage paid in Louisville, KY.
Postmaster: Send address changes to:

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

Copyright

© 1996, The Cobb Group. All rights reserved. *Inside Solaris* is an independent publication of The Cobb Group. The Cobb Group reserves the right, with respect to submissions, to revise, republish, and authorize its readers to use the tips submitted for personal and commercial use. Information furnished in this newsletter is believed to be accurate and reliable; however, no responsibility is assumed for inaccuracies or for the information's use.

The Cobb Group and its logo are registered trademarks of Ziff-Davis Publishing Company. *Inside Solaris* is a trademark of Ziff-Davis Publishing Company. Sun, Sun Microsystems, the Sun logo, SunSoft, the SunSoft logo, Solaris, SunOS, SunInstall, OpenBoot, OpenWindows, DeskSet, ONC, and NFS are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

article "Using tput to Dress Up Your Character-Mode Screens" on page 4. The part that concerns us in this article lies in the technique required to have the user hit only one key without pressing [Enter]. The solution to our problem lies in two little-used UNIX commands in the Sun/Solaris world—`stty` and `dd`.

The GetUserKeystroke function

We'll solve our problem by creating a Bourne shell function named `GetUserKeystroke` that will read and return one character from the user's keyboard. This function returns the character to the calling program, so the calling program knows exactly which character the user typed. (Please note that the `GetUserKeystroke` function will also work with the Korn shell.)

To achieve this functionality, we need to follow these steps inside our function:

1. Save the current terminal settings.
2. Use `stty` to put the terminal into raw mode so we can read one character from the keyboard at a time.
3. Use `stty` to disable echoing of characters.
4. Use `dd` to read one character from the end user.
5. Return that character to our calling program.
6. Restore the terminal to its original settings.

We'll also perform one additional step, as a convenience to the user: We'll display the character the user typed on the terminal screen.

Figure B shows the code for the `GetUserKeystroke` function. As you can see,

the function contains only six lines of code, but it's a very powerful six lines indeed.

Preserve the terminal's current state

In the first line of the function, we use the `stty -g` command to dump all the current `stty` settings of the user's terminal. We do this so that we can restore the terminal to its original state at the end of our function. If you execute the `stty -g` command on your terminal, the output will look something like this:

```
2506:5:d04bd:8a3b:3:1c:8:15:4:0:0:0:
➔11:13:1a:19:12:f:17:16:78:f4
```

This line of information describes the state of the terminal device in a format that `stty` can read. It doesn't matter what kind of terminal you're using. You could be using a VT-100, an X terminal, an `xterm` window, a `shelltool` terminal window, or any other terminal device, and the `stty -g` command will print all the pertinent information in a form that it can read back again.

We execute the `stty -g` command inside a pair of grave accent (```) characters: This tells the shell script to retrieve the command's output and put it in the `oldTtySettings` variable.

Figure B

```
GetUserKeystroke ()
{
  # Save old tty settings.
  1 oldTtySettings="`stty -g`"

  # Set echo off and raw mode on
  2 stty -echo raw

  # Get the keystroke from the user
  3 userKeystroke="`dd bs=1 count=1 2> /dev/null`"

  # Return character to calling program
  4 echo "$userKeystroke"

  # Display character on the screen
  5 echo "$userKeystroke\c" >&2

  # Restore the previous tty settings
  6 stty $oldTtySettings
}
```

This function, `GetUserKeystroke`, allows us to read a single keystroke from the user's terminal.

Change the terminal's state

Now that we've saved the terminal's current state, it's time to change it. In the second line of this function, we use the `stty` command again. This time, it does two tasks for us. First, we use it to disable echoing of characters to the screen, via the `-echo` parameter. Second, we use it to put the terminal in raw mode via the `raw` parameter.

Disabling the echoing of characters means that no matter what the user types, Solaris will not display them on screen. Programs often do this when a user needs to enter a password—the user never sees the password displayed on the screen.

Putting the terminal in raw mode enables us to read one character at a time from the user. Normally, Solaris buffers each character you type until you press the [Enter] key and processes the command keys, such as kill, erase, rprnt, etc. In raw mode, the characters coming in from the keyboard aren't held until you press the [Enter] key. They're available to an application the moment you press them. Putting the terminal in raw mode also disables the processing of all the command keys.

Read a character from the terminal

With the terminal in raw mode, we can read the user's keystrokes one character at a time. Unfortunately, the `read` command built into the Bourne/Korn shells normally used to read user input just doesn't work this way—we need another software tool. We need a command that can read one character at a time. Fortunately, the UNIX `dd` command provides the functionality we need.

By default, `dd` reads from the standard input stream and writes to the standard output stream. We take advantage of this by reading one character from the standard input stream (the user's keyboard, in this case) and writing that character to the standard output stream.

We've supplied two arguments to the `dd` command. The first, `bs=1`, tells `dd` that we're going to read and write with a block size of one byte. The second, `count=1`, tells `dd` how many blocks to read at a time. This combination of parameters tells `dd` to read a single byte and print it to the output. We also redirect the standard error stream to `/dev/null` so we don't have to see any status messages on the screen. Since we've executed the `dd` command inside grave accents (```), the shell takes the output of the `dd` command, in this case the character the user typed, and assigns it to the variable `userKeystroke`.

Return the character to the caller

After we read the character the user typed, we need to return it to the caller. We do this by printing it to the standard output device. This necessitates a somewhat unusual calling convention for our shell function. Rather than calling it in the traditional fashion, we call it by enclosing it in grave accents. Then the caller must assign the result to a variable, as we did when we called the `dd` command previously. We'll give a demonstration later, in the section How to Use `GetUserKeyStroke`.

Display the character on the screen

The fifth line of our function sends the character that the user typed to the standard error stream. Unless the calling function redirects the standard error stream somewhere, this line prints the character to the screen. We use the `\c` symbol so that the `echo` statement doesn't display a carriage return after the character.

Restore the terminal's state

Now we've done everything we wanted. We've read a single character from the user, displayed it, and returned it to the caller. Now all we need to do is clean up after ourselves. Since we altered the terminal's state with `stty`, we need to restore it.

The last line in our function restores the `tty` device (e.g., your terminal or shell tool) to the settings that were in effect before the script called `GetUserKeyStroke`. For most situations, this means to take the terminal out of raw mode and re-enable character echoing.

How to use GetUserKeyStroke

When you want to use `GetUserKeyStroke` in your scripts, you must call it like this:

```
userChoice=`GetUserKeyStroke`
```

This varies from the traditional method of calling functions, and it gets you to think differently about invoking function calls. By calling the `GetUserKeyStroke` function within the grave accent characters, the shell executes the function and places the standard output of the function into the variable `userChoice`. We have to do it this way because the `GetUserKeyStroke` function returns a value by printing the return value to the standard output stream.

The code fragment shown on the next page gives a simple example of how you can use the `GetUserKeyStroke` function. As you can see, we

prompt the user for Y to continue or Q to quit. Next, we read the user's key press into the variable `userChar` using the `GetUserKeystroke` function. Finally, we decide what to do based on the keystroke. If it's a y or Y, we print the word *Continuing*, and execute the loop again. If it's a q or Q, we print *Thanks for playing!* and stop the script. Otherwise, we print *?What?* and prompt the user again for a Y or Q.

```
while true
do
    echo "Press (Y) to continue, (Q) to quit: \c"
    userChar=`GetUserKeystroke`
    echo
    case $userChar in
        y|Y) echo "Continuing...";;
        q|Q) echo "Thanks for playing!"
            exit;;
        *) echo "?What?";;
    esac
done
```

Special note

After studying the `GetUserKeystroke` function, you may be wondering why we disable character echoing when we intend to display the character on the screen anyway. We do so because terminal echoing doesn't let you control *when* the display occurs. If we control it, we'll display the character only when it's convenient for us.

If we allowed the display to happen at any time, then if the user pressed a key while displaying a menu, it could disrupt the appearance of the menu. In fact, since the terminal uses special key sequences for some operations, an inappropriate keystroke could render the screen totally unreadable.

Conclusion

In retrospect, `GetUserKeystroke` appears to be a relatively simple six-line function that improves your shell programs. However, as the saying goes: Hindsight is always 20/20, and many veteran shell programmers have learned a few things from being exposed to this code.

In a nutshell, `GetUserKeystroke` solves a particular programming problem, but more than that, it provides insight into several important UNIX shell programming topics. This function shows how to use the `stty` command to control terminal characteristics, gives an unusual example of the `dd` command, stretches the limits of I/O redirection to return a variable from a function, and displays a variable to the screen on the standard error device.

Once you've mastered the technique presented in this article, you'll find that it opens your mind to other possible applications. For instance, you can use this technique to create a `GetUserPassword` function that displays an asterisk for each character the user types in after you prompt the user for a password. ❖

SCRIPTING TECHNIQUES

Using tput to dress up your character-mode screens

By Al Alexander

One of the most enjoyable things in Solaris is shell programming. It's very satisfying to create shell programs to help people accomplish their tasks more easily. It's even better when you're able to go the extra mile and give your programs a great look.

While it's very simple to make your shell scripts have simple displays, it's much nicer to dress them up and make your programs appear more professional. There is a standard

UNIX command called `tput` that allows you to do many fancy things to improve the appearance of your shell programs, including making text bold, underlining text, making it blink, and even controlling cursor position.

In this article, we'll explore the `tput` command and show how you can use it to liven up the output of your shell programs. We'll create a simple menu screen that shows how you can use `tput` to clear the screen and display underlining and bold effects on your terminal.

The basics of tput

The `tput` command uses the terminfo database to make certain terminal-dependent characteristics available to command-line users and shell programmers. It's important to understand that your program can tell a terminal to underline a certain character, but if the terminal doesn't provide the capability for underlining text, you won't get underlines on your screen.

The terminfo database contains entries for many common terminal types. Each termcap entry has a list of capabilities that the terminal supports. Table A describes the capabilities that we use most often with the `tput` command, along with the standard name (Cap-name) for the capability. If you want to see a comprehensive list of device capabilities available for use with `tput`, you should read the `man` page for terminfo.

Table A

Cap-name	Desired effect
<code>clear</code>	Clear the screen
<code>smso</code>	Write any following characters in standout mode
<code>rmso</code>	End standout mode
<code>smul</code>	Start underlining mode
<code>rmul</code>	End underlining mode
<code>cols</code>	Return the number of columns on the terminal
<code>cup r c</code>	Move the cursor to a specific location

These are some of the most commonly used terminal capabilities available in the termcap database.

Most terminal capability entries in the terminfo database describe the special character sequences you must print to do some special operation on the terminal. Other entries describe the key sequences the terminal returns when you press one of the special keys on the keyboard, such as [F5].

Unfortunately, you can't access the terminfo database directly with a shell script. Normally, a program must examine the database to find out what to do. Fortunately, Solaris provides the `tput` command, which simply looks up the specified Cap-name and prints its value to the standard output stream. To do so, you simply need to execute the command

```
tput Cap-name
```

replacing *Cap-name* with the name of the terminal capability you want to invoke. For example, Figure A shows a standard CDE terminal screen where we put the terminal in standout mode, add underline mode, and then turn off the modes in the same order. As you can see, the CDE terminal window supports this behavior.

Unfortunately, you can't rely on all terminals being able to allow you to turn on and off modes in any arbitrary order. In general, you should use modes sparingly, and you should probably avoid mixing them whenever possible. This will maximize your chances of having your screen look the way you intend it to when the user runs your script on a new terminal.

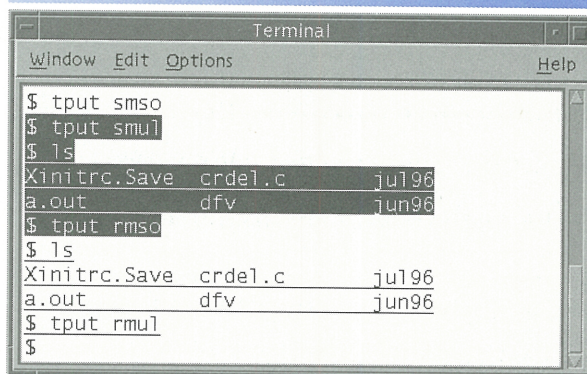
As an example, you might want to execute the commands shown in Figure A on an OpenWindows terminal window. If you do so, you'll notice that when we turn off the standout mode, the OpenWindows terminal loses the underline mode as well.

There's a special case of `tput` that you need to be aware of. Some terminal capabilities require one or more parameters in order to execute properly. For example, the `cup` capability, which moves the cursor, requires that you provide the desired cursor position as two numbers, the desired row and column. The `tput` command accepts these parameters after the Cap-name. Thus, to move the cursor to the tenth column on row four, you'd type

```
# tput cup 4 10
```

We often use another Solaris command, `infocmp`, to determine what characteristics a given terminal type supports. You can use the `infocmp` command to extract specific terminal

Figure A



The CDE terminal window supports turning modes on and off in any order.

information from the terminfo database. For instance, if you want to know what display characteristics your current terminal supports, simply type

```
# infocmp
```

This command tells you all the capabilities your current terminal supports. For a large, complex terminal, `infocmp` is likely to provide

many lines of information. (The CDE terminal window, in fact, returns 30 lines of terminal capability information.)

If you're writing a script for a user who uses a terminal you don't have access to, all is not lost. You can use `infocmp` to tell you what terminal capabilities the user's terminal supports, as long as the terminal is in the terminfo database. You can do so by specifying the desired terminal type on the command line. For example, if you want to determine the display characteristics of a VT-100 terminal, you can type the command

```
# infocmp vt100
```

Within shell programs, we often create shell variables that turn on and off various display capabilities. [Figure B](#) shows a sample menu screen that you might create for your users. [Figure C](#) shows the shell script that displays the menu shown in [Figure B](#). The five colored lines show the definitions of five shell variables that control various display attributes.

Once you create these terminal-control variables in this manner, they contain the terminal-dependent codes necessary to turn on or off the bold and underline features on your current terminal. This technique makes your programs more efficient, because your scripts don't have to execute the `tput` command each time you want to change the terminal attribute.

Please note the use of the braces in the `echo` statements. This is the method that the Bourne and Korn shells use to separate your shell variables from other text when it's necessary to run them together. Suppose we had the statement

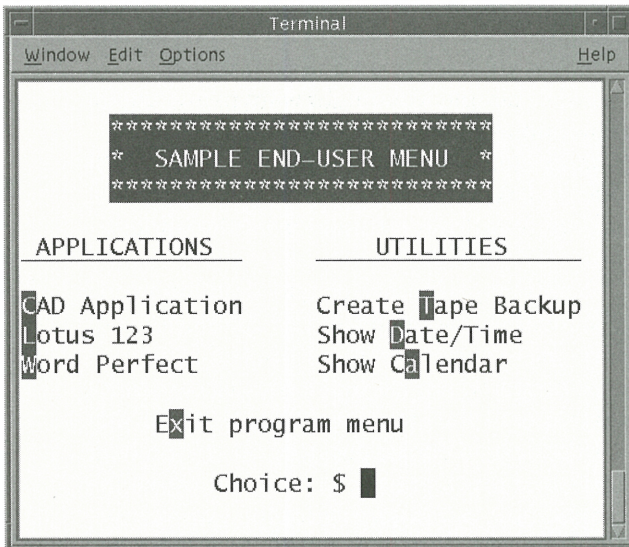
```
echo "Play the ${B}x${Bx}ylophone"
```

Normally, this would display the `x` in bold mode. However, if we didn't use the braces, the statement would look like this:

```
echo "Play the $Bx$Bxylophone"
```

This statement wouldn't execute correctly, because it's telling the shell to print the value of the variables `Bx` and `Bxylophone`. The variable `Bx` turns off bold mode, but since we never enabled it properly, it doesn't print anything. Similarly, since we don't have a variable named `Bxylophone`, it won't print anything either. If you execute this line, you'll simply see the string `Play the` on your screen.

Figure B



Using the `tput` command, it's easy to create professional-looking menus for your shell scripts.

Figure C

```
#!/bin/sh
B='tput smso' # Start bold mode
Bx='tput rmso' # End bold mode
C='tput clear' # Clear screen
U='tput smul' # Start underline mode
Ux='tput rmul' # End underline mode

echo "${C}
${B}*****${Bx}
${B}* SAMPLE END-USER MENU *${Bx}
${B}*****${Bx}

${U} APPLICATIONS  ${Ux}      ${U} UTILITIES  ${Ux}

${B}C${Bx}AD Application      Create ${B}T${Bx}ape
Backup
${B}L${Bx}otus 123              Show ${B}D${Bx}ate/Time
${B}W${Bx}ord Perfect          Show C${B}a${Bx}lendar

E${B}x${Bx}it program menu

Choice: \c"
```

This script displays the sample menu shown in [Figure B](#).

Conclusion

The `tput` command can greatly enhance the appearance of your shell programs, which makes them easier to use. If you create shell variables that turn these modes on and off, your program won't have to call `tput` each time you want to change the terminal mode. These techniques will help you create great-looking shell scripts in a very short time. ❖

Alvin J. Alexander is an independent consultant specializing in UNIX and the Internet. He has worked on UNIX networks to support the Space Shuttle, international clients, and various Internet service providers. He has provided UNIX and Internet training to over 400 clients in the last three years.

SHELL PROGRAMMING

Make a shell script mail you a summary

If you often run shell scripts unattended, such as in the background, or via the `at` command, you may find that it would be helpful if you could get a summary of their operation. In this article, we'll show you how to make a shell script mail you a summary. Using this technique, you'll be able to run your shell scripts, content in the knowledge that when you want to check the results, you can simply look at your mail.

Sending a message with mail

As you probably know, you can send mail to someone by using the `mail` command. To do so, you give the `mail` command the recipient list on the command line, and then it accepts a mail message via standard input. Once it finds the end of the input stream, it mails the text to the specified recipients.

Therefore, to have a shell mail us a summary report, we must provide two things: an appropriate recipient list and a message body. Fortunately, both of these are easy to obtain. We'll use your user name as the recipient list and the output of your command list as the message body.

The recipient list

Who is interested in the results of your script file? Presumably, only you. Therefore, in your script file, you can send a message to yourself with the command

```
$ mail myname
```

where *myname* is your user name.

However, this technique has one slight disadvantage. If you give a copy of your script to others, they must edit it to send the results to their mailboxes or you'll get their reports.

Fortunately, there's a better solution. Rather than hard-code your user name into the recipient list, we'll extract your user name from the environment. If you've ever executed the `set` command and looked at all the environment variables, you may have noticed the one named `LOGNAME`, which the shell set to your user name. All you need to do in your script is use the `LOGNAME` environment variable as the recipient, so whoever runs the script gets the report. You can do so like this:

```
$ mail ${LOGNAME}
```

The message body

Now that we've properly addressed our summary report, all we need to do is send the `mail` command the body of the message. If your report merely consists of a mention that the script has run, you can get away with something simple like this:

```
mail ${LOGNAME} <<!
The test script executed normally.
!
```

We use a here document (described in the article "Automating Applications that Accept User Input" in last month's issue) to feed a message body to the `mail` command. One problem with this technique is that you have to know all the reports that you want to send at the time you run your script. That's usually not the case.

The normal case is that you'll want to see all the output of your script in the mail message. That way, you can examine the output for anything unexpected. As you probably suspect, if you have a single command, you can simply pipe its output to the `mail` command. For example, if you want to find a list of all the files

named *core* in your home directory and below, you might use the command

```
$ find ~ -name core -print | mail ${LOGNAME}
```

If you have multiple commands to execute in your script, what do you do? You could try creating a temporary file and appending the output of each command to it. Then, after you mail the results to yourself, you can remove the temporary file. For example, suppose you want a listing of the contents of your *.dt/Trash* and *.wastebasket* directories. You could do it like this:

```
$ ls ~/.dt/Trash >/tmp/test
$ ls ~/.wastebasket >>/tmp/test
$ mail ${LOGNAME} </tmp/test
$ rm /tmp/test
```

The first line creates the file */tmp/test* and fills it with the list of files in your *.dt/Trash* directory. The second line appends the listing of your *.wastebasket* directory to it. The third line mails the resulting file to you and the fourth line deletes the temporary file.

However, this technique has two disadvantages: First, you must create a temporary file that won't be used by some other process. Second, you must be sure to clean up the file when you're finished.

Another way you can find the list of files is to create a shell script that contains the list of files you want to execute. Then you can redirect the output of that shell script to the *mail* command. For our example, we could create a shell script that does both directory operations, like this:

```
ls ~/.dt/Trash
ls ~/.wastebasket
```

Assuming our shell script is named *test*, we can execute it like this:

```
$ test | mail ${LOGNAME}
```

This technique works very well. However, for very simple jobs, it may be too much work. A good method for small command lists is to tell the shell to execute your list of commands in a subshell. Then you can pipe the output of your subshell to the *mail* command. In effect, it's the same as the technique we just showed you, but you don't need to explicitly create a shell script in a file.

All you need to do with this technique is to put your command list in parentheses, with

the pipe symbol outside the parentheses. You can separate your commands with semicolons or newlines. You can process the same example two ways:

```
$ (ls /; ls ~) | mail ${LOGNAME}
```

or

```
$ (ls /
> ls ~) | mail ${LOGNAME}
```

Capturing error output

So far, we've only shown you how to mail the standard output of a command to yourself. Once you try this technique, you may find that parts of the output you're used to seeing on the console don't arrive in your mail message. This happens because many programs print normal results on the standard output stream, also known as *stdout*, and errors on the standard error stream, known as *stderr*.

You may be interested in both the normal output of your commands *and* the error output. If this is the case, then you'll have to know which shell you're using when you execute your script. If you're using the C shell, you can pipe both the *stdout* and *stderr* output to the *mail* command like this:

```
$ cmd !& mail ${LOGNAME}
```

As you can see, all we did was use the *!&* operator, which tells the C shell to pipe the *stderr* stream along with the *stdout* stream. If you're using the Bourne or Korn shell, it's a little more difficult: These shells don't provide the *!&* operator.

However, you can do the job by telling the shell to redirect the *stderr* stream to the same place as *stdout*. Then you can pipe the resulting output to the *mail* command.

You can merge the *stderr* and *stdout* streams by using the I/O redirection command *2>&1*, which tells the shell to take the output of stream 2 and send it to the same place as stream 1. Since stream 2 is the *stderr* stream and stream 1 is the *stdout* stream, this command does exactly what we want.

Now we're ready to put it all together. Please note that we must use the I/O redirection command that merges the standard error stream to the standard output stream *before* the pipe symbol, because the pipe symbol separates different sections of the command. If we put the I/O command after the pipe symbol, we'd be telling the shell to take any error output

from `mail` and send it to the standard output. Our completed command line is

```
$ cmd 2>&1 | mail ${LOGNAME}
```

Conclusion

If you're as busy as most people, then you may find this trick to be a time-saver. It's nice to be

able to run your scripts and programs confident in the ability to examine the results at your leisure rather than watching for the output before it scrolls off the screen. One added benefit is that since the summary is in your mailbox, rather than just on your screen, you can elect to print it, forward a copy to someone else, or just ignore it until you need to look at it again. ❖

SYSTEM ADMINISTRATION

Don't clutter your system with core images!

As you may know, sometimes when a program crashes, it creates a large file named *core* in the current directory. Your program may also create a *core* file if you press the quit key (usually defined as [Ctrl]\) while it's running. If your program doesn't explicitly trap the quit signal, Solaris will write the *core* file and terminate the program.

What is the core file?

The *core* file is simply a file that contains the program code and all the data that the program is using. Application developers often use the *core* file to examine various bits of data that the application was using to see why it crashed. Since some programs are quite large, and they use a lot of data, these *core* image files can consume quite a bit of space.

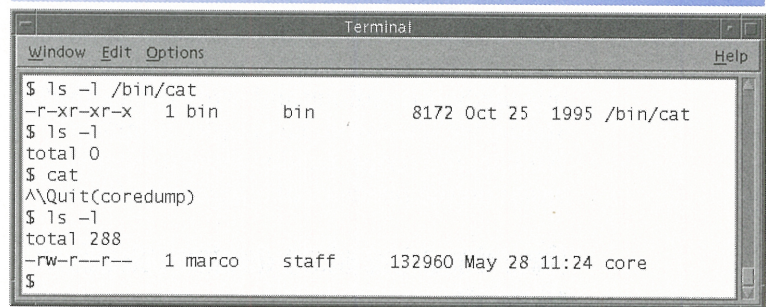
Even smaller programs can generate a surprisingly large *core* file. For example, Figure A shows that the program `cat` generates a *core* image file of over 100K. This is rather surprising for a program that's a mere 8,172 bytes long in Solaris x86 v2.5.

Why is the *core* file so large for `cat`? It's certainly not from the amount of data that the `cat` command uses. It turns out that many Solaris commands, to save space and ease program maintenance, access a shared library of code when they execute. The `cat` command is one of these. Thus, when you run `cat`, it loads some code from a shared library to perform some tasks for it, such as file manipulation.

The ulimit command

If you're not diligent about removing unnecessary *core* files, you may find a large amount

Figure A



```
Terminal
Window Edit Options Help
$ ls -l /bin/cat
-r-xr-xr-x 1 bin bin 8172 Oct 25 1995 /bin/cat
$ ls -l
total 0
$ cat
^\\Quit(coredump)
$ ls -l
total 288
-rw-r--r-- 1 marco staff 132960 May 28 11:24 core
$
```

Here we create a core file by terminating the `cat` command with the quit key.

of disk space consumed with these useless files. Wouldn't it be a lot simpler if you could just tell Solaris that you're not interested in *core* files? Then you'd be spared the bother of cleaning them up.

It's a simple task to tell Solaris not to generate these *core* files. The `ulimit` command allows you to set limits on resource usage by various users. One of the limits you may set is the size of the *core* image file that an application can generate.

If you want to prevent a *core* file from being generated at all, you can simply use the command

```
$ ulimit -c 0
```

which tells Solaris to limit the size of any *core* image files to 0 blocks. This way, Solaris won't allow your applications to create *core* image files.

Make sure everyone runs ulimit

Unfortunately, when you run the `ulimit` command, you're telling Solaris to prevent *you* from

generating *core* files. All the other users on the system can still generate them. If you want to prevent everyone from creating *core* files, you need to ensure that everyone runs the `ulimit`

command each time they log in. The article "Execute Commands When Logging In," in the April issue, describes a procedure you can use to make everyone execute this command. ❖

SCRIPTING TECHNIQUES

Conditionally executing shell scripts

You sometimes might want to run a shell script at a scheduled time to perform some administrative tasks, like making a backup. If you use `crontab` to schedule your scripts, you're just about set. The only problem we encounter with this technique is that while you normally want to execute the script, there are occasions when you don't.

You could remove the appropriate `crontab` entry when you don't want to execute the script and then reenter it when you do. However, this method is tedious and error-prone. In this article, we'll show you a simpler technique: We'll let the script decide when to execute based on the presence of a particular file.

How does the shell decide whether to run?

First we need a method to allow the shell script to decide whether to run. It turns out that this is very easy to do. Figure A shows our demonstration shell script, named *test*. The highlighted lines show the part that checks for the file `/locks/test`.

The heart of the trick is the `-e filename` clause in the condition section of the `if` state-

ment. This clause is true if *filename* specifies a file that exists; otherwise the clause is false. The rest of your shell script remains unchanged.

In our demonstration, if the file `/locks/test` exists, the script prints a message to let you know why it terminated. Then it executes the `exit` statement to stop processing the script.

Create a directory of lock files

In order to make this technique manageable, you should keep your lock files in a centralized location. For our purposes, we simply created the `/locks` directory. Creating this directory gives you the ability to easily determine which shell scripts are locked out on your system. All you need to do is execute the command

```
$ ls /locks
```

and you'll see a list of the scripts that are currently locked out from use.

Whenever you want prevent a shell script from operating, just execute the command

```
$ touch /locks/scriptname
```

where *scriptname* is the name of the script you want to lock out. Then, when you want to

Figure A

```
#!/bin/ksh

# Test whether shell script is allowed to run

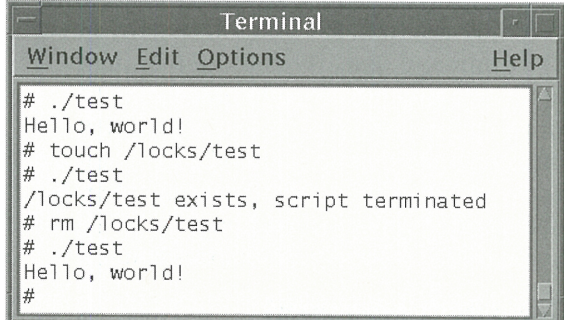
if [ -e /locks/test ]; then
    echo "/locks/test exists, script terminated"
    exit
fi

# Normal shell script operation starts here

echo "Hello, world!"
```

If this shell script finds the file `/locks/test`, it prints a warning and exits.

Figure B



```
Terminal
Window Edit Options Help

# ./test
Hello, world!
# touch /locks/test
# ./test
/locks/test exists, script terminated
# rm /locks/test
# ./test
Hello, world!
#
```

Here's how the technique works: First *test* runs normally, we lock it out, and then we allow normal operation again.

allow access to this script again, just remove the lock file with the `rm` command

```
$ rm /locks/scriptname
```

Figure B shows our demonstration script in action. First we run `test` normally, and then we lock it out by creating the file `/locks/test` and run `test` again. Finally, we delete the lock file and `test` runs normally again.

Conclusion

This technique is useful whenever you want to prevent a certain script file from running. If you're scheduling a late night and don't want to be slowed down by a large finite element analysis job, you can turn off the script that runs it. Or perhaps your tape drive was removed for repair. If so, you can use this technique to turn off the backup script. ❖

FILE MANAGEMENT

Hard and soft file links

By Marco C. Mason

As you're aware, your file system greatly resembles a tree structure, like the one shown in Figure A. From the root directory, `/`, everything grows upwards. You graft other file systems onto the root file system at various places to add more disk space to your machine or to allow simple access to files on other machines or file systems.

However, there are cases where a simple tree structure isn't the best choice. Sometimes you want to provide different users access to a restricted access file without adding more groups to the system. Or maybe you want to have a command with multiple names, but you don't want to fill your disk drive with multiple copies of the command. Perhaps you simply want to effectively move files to a more logical location but still allow users to pretend that the files exist at their old location.

In cases like these, you'll want the ability to graft together some of the branches of your tree, as shown in Figure B. The blue part of the file system is on a different file system than the root. When the root file system ran out of room, we moved the `SUNWss` directory from the root file system to the export file system. We then deleted the `SUNWss` directory from the root partition and created a link to the `SUNWss` directory on the export file system. Now, none of the users of the `SUNWss` directory will notice that the file moved.

Directory entries and inodes

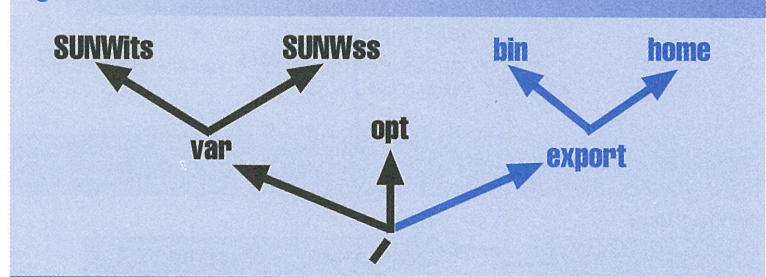
When you're working with files, you're actually working with directory entries. The directory entry keeps track of two basic pieces of

information about the file: its name and the inode number that describes the file.

All the basic information that describes the file is stored in the inode. You'll find the permissions, user ID of the owner, last access time of the file, and much more in the inode.

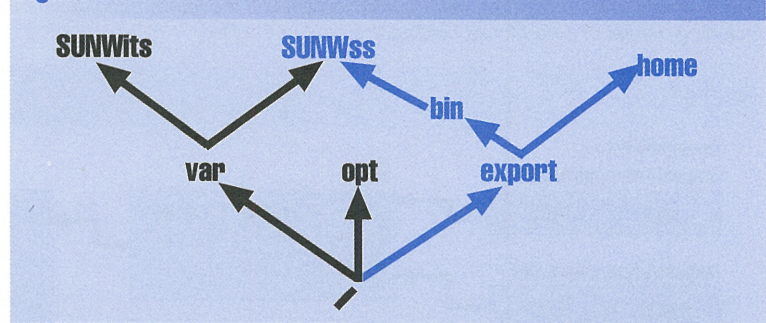
The two most interesting fields in the inode are the block list and link count fields. The block list keeps track of all the blocks of storage on the disk that are part of the file. The link count field tells how many directory entries refer

Figure A



The traditional view of a file system is a tree of files and directories growing from the root directory.

Figure B



File system maintenance is easier when you can merge some branches in the tree.

to the inode. [Figure C](#) shows how the directory entry, inode, and file data are connected.

Hard links

From the existence of the link count field, you may infer that multiple directory entries may refer to the same inode. If you think that, you're correct. A hard link is simply the directory entry that refers to the inode that describes the file.

When you create a new directory entry that refers to this inode, UNIX increases the link count field in the inode. Similarly, when you delete a directory entry that refers to this inode, the link count decreases. If the count decreases to 0, then UNIX deletes the file specified by the inode.

For example, suppose we're in the directory `/export/home/marco`, and we create two subdirectories, `abc` and `def`.

```
$ cd /export/home/marco
$ mkdir abc
$ mkdir def
```

Then let's create a file named `test` in the `abc` directory and list the directory `abc`.

```
$ ls >abc/test
$ ls -l abc
-rw-r--r-- 1 marco  staff    69 May 12
↳12:24 test
```

Now let's get ahead of ourselves and create a hard link to the file in our `def` directory, calling it `temp`. Then we'll list the directory `def`.

```
$ ln abc/test def/temp
$ ls -l def
-rw-r--r-- 2 marco  staff    69 May 12
↳12:24 temp
```

As you can see, both directories are identical, except that the number in the second column changed between directory listings and the filenames are different. The number in the second column is the link count field. It is 1 for the first directory listing and 2 for the second directory listing because we added a link between the two `ls` commands. If you go back and list the `abc` directory, you'll see that it now has a 2 in the second column.

At this point, our directory entries and inode look something like [Figure D](#). Next, let's delete the file `abc/test` and then list the `def` directory again.

```
$ rm abc/test
$ ls -l def
-rw-r--r-- 1 marco  staff    69 May 12
↳12:24 test
```

Now the link count goes back to 1. Even though we created the file in the `abc` directory and then deleted it, the file exists and is currently linked to the `def` directory. UNIX doesn't care which directory you link a file to first. It deletes a file only when the link count goes to 0. If we delete the file from `def`, the link count will go to 0, and UNIX will free all the space currently allocated to the file.

Creating a hard link is very easy. All you need to do is use the `ln` command like this

```
$ ln source dest
```

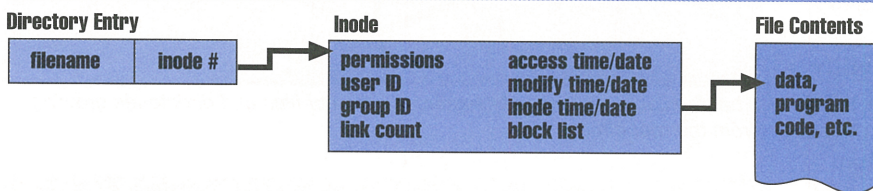
where `source` is the name of the file you want to link to and `dest` is the name you want for the link. The `dest` field is optional, and when you omit it, it defaults to the same name as `source`, but in the current directory.

Let's suppose your friend Joe-Bob created a program named `smorgasbord-selector` that chooses the restaurant you'll go to for lunch. You might think that's a bit too much to type, so you want to name your own copy `lunch`. Rather than copy the entire program, you can just create a link to it with the command

```
$ ln /export/home/joe-bob/smorgasbord-
↳selector lunch
```

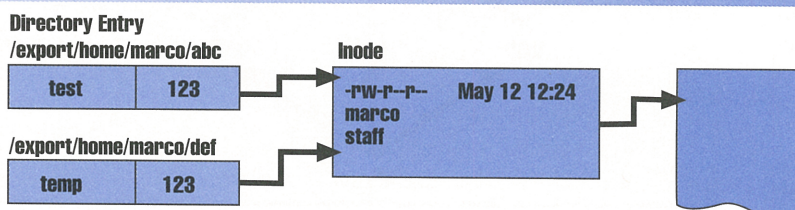
One advantage of using a hard link rather than a copy is that you're expending only the space for a single copy of the program. Another benefit is that if your friend updates the program with more

Figure C



The directory entry refers to an inode that contains all the information describing a file, including the list of data blocks holding the data in the file.

Figure D



At this point, we have two directory entries pointing to our test file.

restaurants, the next time you run `lunch`, you'll have access to the new restaurants.

Hard links have their disadvantages as well. First, you can't hard link one directory to another because the `.` and `..` entries in a directory link to the current and parent directories, respectively. Solaris prevents you from linking to directories so that your directory links don't get confused with Solaris'.

To illustrate why you can't create hard links to directories, let's assume that you can. Suppose you create the directory `/export/home/fred/games`. When you do so, Solaris automatically creates the `..` entry in `games` to refer to `/export/home/fred`. Now suppose Fred's buddy George started work at the same company and offered George the use of his games. George created a link to Fred's `games` directory, just as you might expect.

Now, George is in his home directory, `/export/home/george`, and wants to play a game. He types `cd games` to get to the `games` directory, plays `tic-tac-toe`, and then wants to get back to work. He types `cd ..`, and where is he? He's in Fred's home directory!

Suppose Fred leaves the company, and the system administrator removes Fred's account. George would still have access to Fred's `games` directory. But now when George types `cd ..` from the `games` directory, UNIX would get confused. The directory that the `..` directory entry refers to no longer exists!

Another shortcoming with hard links results from the fact that the directory entry directly specifies an inode number. Because of this, you can't make a hard link refer to a file on another file system. The reason for this is twofold: First, there's no data item in the directory entry to specify on which file system the inode resides. The second reason is the most important: It prevents file system corruption.

Suppose for a moment that you *could* access an inode in a different file system, and the file system happens to be a directory tree on another machine on the network. If you shut down the network and the user on the remote machine continues working, it's possible that the inode your directory entry refers to could be reassigned to some other file. Once the network comes up, if you were to write to the file specified by your link, you could damage a file on the other person's machine.

Soft, or symbolic, links

In Figure A, we showed you a diagram that violates *both* limitations of hard links. Not only does it show a link to a directory, but it

shows a link across file systems. While a hard link has both of these limitations, a symbolic link has neither.

A hard link works at the very bottom level of the file system to allow direct access to an inode and, therefore, operates very quickly. A symbolic link does its magic by operating at a higher level than a hard link. Rather than storing the inode number of a file, it stores the file reference in a symbolic form: text.

When UNIX uses a symbolic link, it goes through an extra level of processing. When it finds itself at a symbolic link, it reads the symbolic link to get the name of the file or directory being linked to. Then it starts all over looking for the file.

For example, suppose you have a file named `/alpha/beta`, and you have a symbolic link to it called `/one/two`. If you attempt to read file `/one/two`, UNIX first locates the directory entry for `/one/two`. Next it reads the symbolic link information and finds out that the file's real name is `/alpha/beta`. Then UNIX starts all over and looks for `/alpha/beta`.

The chain could continue, where `/alpha/beta` refers to yet another file. As you can see, UNIX has to do more work to resolve a symbolic link. However, since it's symbolic, the link can refer to any directory, even one on another machine, as long as it's mounted and accessible in your machine's file system hierarchy. If a symbolic link resolves to a file that doesn't exist, UNIX will simply generate a *file not found* error.

UNIX allows you to create symbolic links to a directory because the strange situation that could happen with hard links can't happen with a symbolic link. Since you can't delete a directory's parent directory, UNIX will always be able to process a `cd ..` command without error. In fact, to make operating with symbolic links less confusing, the shells have support built in to help change directories with less confusion. So if you change to a directory through a symbolic link, you'll get to your original location when you execute the `cd ..` command.

For example, with a default installation of Solaris, the `/lib` directory is really a symbolic link to the `/usr/lib` directory. So if you're in the root and type `cd lib`, you'll be in `/usr/lib`. However, the `pwd` command will report that you're in `/lib`, and `cd ..` will take you back to the root.

You can create a symbolic link by adding the `-s` option to the `ln` command. Thus, you can create a symbolic link like this

```
$ ln -s source dest
```


where *source* is the name of the file or directory you want to link to and *dest* is the name of the symbolic link. Please note that unlike a hard link, *source* need not exist at the time you create your link.

You can see symbolic links when you issue an `ls -l` command. Whenever you encounter a symbolic link, Solaris prints the standard directory line and then appends `-> source`. Thus, if you're in the root directory, and you type `ls -l`, the line for the *lib* directory appears as

```
lrwxrwxrwx 10 root  sys      512 Apr  1
->20:50 lib -> ./usr/lib
```

As you can see, it also prints an *l* as the file type in the permissions mask. No matter whether the file is a file or a directory, you'll see an *l* as the first character.

Please note something else about the permissions mask. Notice that the permissions are `rw-rw-rwx`, giving you, your group, and

the rest of the world complete access. This isn't bad as it seems at first glance. It's not really telling you that just anyone can erase all the files in the *lib* directory.

Symbolic links have one additional feature: You can set a priority mask for them that can further restrict access to a file or directory. Rather than using the permissions for the symbolic link as the permissions for the file or directory the link refers to, the permissions simply tell Solaris which permissions are allowed *if they're allowed on the original!* If you've removed any permissions from either the original file or the symbolic link, then Solaris won't grant the permissions.

On a default installation of Solaris, the permissions on the `/usr/lib` directory are `rw-rw-rwx`. Therefore, even though the symbolic link grants write access to the world, the `/usr/lib` entry restricts it, so no one outside the root's owner and group may write to this directory. ❖

QUICK TIP

Build a keyword index for man

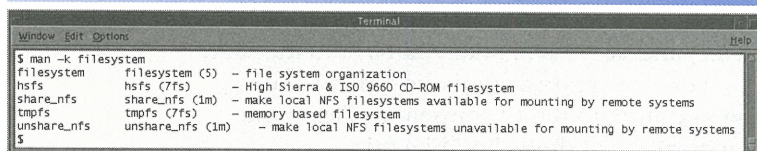
As you know, if you want to read a `man` page, you have to know the name of the command, and optionally the section number. What you may not know is that `man` will help you find the appropriate manual sections if you don't know exactly what you want. All you need to know is a word that's related to what you want to find. You use this word in the `man` command like this: `man -k keyword`, where *keyword* is a word relating to the information you're seeking.

As an example, Figure A shows the results of running the command `man -k filesystem`. As you can see, `man` found five `man` pages containing the keyword `filesystem`. The reason it didn't find the obvious candidates `mount` and `umount` are because these `man` pages use the phrase `file system` rather than the word `file-`

system. As you can see, this isn't a perfect method. However, it does give you a good way to start dredging the `man` pages for information.

However, before you can use the `-k` option with `man`, you must build the keyword indices. This is where `catman` comes into play: It's the program that builds the keyword indices for the `man` pages.

Figure A



Here's what `man` turned up when we searched for the keyword `filesystem`.

Table A

1	User (shell-level) commands
2	UNIX system calls
3	C library calls
4	File formats
5	Headers, tables, and macros
6	Games and demos
7	Special files
9	DDI and DKI

These are the major categories of standard sections for `man` pages.

All you should need to do is to execute `catman -w` with the list of sections you'd like to index. If you omit the section list, `catman` assumes that you want to index *all* manual sections. [Table A](#) shows the standard sections used to organize `man` pages. For example, to build a keyword index of sections 1 through 4, just use the command

```
$ catman -w 1 2 3 4
```

You should build the keyword indices only for the sections that you're interested in.

If, for example, you don't do any Device Driver Interface (DDI) programming, then indexing section 9 would be a waste of disk space.

This process of building the keyword index can consume a good bit of disk I/O and CPU time, depending on the number of sections and `man` pages that `catman` must process. Therefore, you may not want to run `catman` during your peak load hours. You could consider this as an example of when you'd want to defer program execution by using the `at` command. ❖

LETTERS

Who's tying up the file system?

I just recently "inherited" a network with several Sun workstations and have a couple of questions. As part of our monthly maintenance schedule, we have to unmount some file systems. Unfortunately, sometimes someone is still using a file on the file system. Right now, I have to use `wall` to tell everyone to get off the file system.

The problem is that someone might be at lunch and won't see the message immediately. Also, some users don't know if they're using the file system. I know there's a better way, but I don't know what it is. Can you help?

Also, my boss wants me to add a dialup line to one of the workstations so some employees can have access after hours. How do I do this?

*Carolyn Belak
Orlando, Florida*

To find out who's using a file system, you can use the two commands `fuser` and `ps`. The first command, `fuser`, tells you which processes are using a file system. To use it, you just type `fuser` followed by the list of file systems you're interested in. The `fuser` command will respond with the list of processes using each file system.

Please note that each process number will have one or more letters following the number. These letters aren't part of the process ID—they're just telling you what the process is doing with the file system. For example, `c` means that the process is using the file system as its current directory, and `o` means that the process has an open file on the file system. You can execute `man fuser` for more information on this command.

Once you've found the list of processes using the file system, you need to convert them to the usernames with the `ps` command. You need to use the `-f` switch to tell `ps` to print a full listing that includes the username. You also need to add the `-p` option, which tells the `ps` command that you're going to specify a list of process IDs that should follow the `-p` command separated by commas, like this:

```
$ ps -f -p1,2,3
```

Now you can tell the specific users to please finish their jobs that use the file system or you can log them out if they don't respond. Suppose that you want to unmount the `/export/oracle` and `/export/remote` directories. Your session might look like this:

```
$ umount /export/oracle
umount: /export/oracle busy
$ umount /export/remote
umount: /export/remote busy
$ fuser /export/oracle /export/remote
/export/oracle: 300c 319co
/export/remote: 301c
$ ps -f -p300,301,319
  UID  PID  PPID  C  STIME   TTY   TIME CMD
  root  300  298   0  09:06:48 pts/3  0:00 /sbin/sh
  root  301  298   0  09:07:01 pts/3  0:00 /sbin/sh
  root  319  300   0  09:10:06 pts/3  0:00 /export/home/marco/
  ↪ thrash
```

As you can see, you can give the `fuser` command a list of file systems, and it will print the file system name before each list of process IDs. Also, when you specify the process ID list to the `ps` command, don't put a space between the `-p` and the list of process IDs.

SunSoft Technical Support
(800) 786-7638

Please include account number from label with any correspondence.

Your second question is much more complex. Obviously, you'll need a data communications line between the remote user and the host. Most people use modems to communicate over telephone lines. The hardware aspect is beyond the scope of this journal, but we've covered the software in past issues.

You'll probably want a PPP link between the remote and host machines. We discussed how to set up the host side of a PPP link in the article "Configuring PPP under Solaris 2.x" in October 1995. The article "Configuring a Remote PPP Dial-up Client for Solaris 2.x" in the February 1996 issue covers the remote user side. ❖

QUICK TIP

Quickly switch between directories

If you're like me, you frequently move between directories in your job. Some directories you use so frequently that you can probably type them without thinking. When the directory names you use are very long, however, typing them can be a tedious and error-prone activity.

It turns out that there's a simple trick you can use to make it much easier. All you need to do is create environment variables for your commonly used directories. Then you can use these environment variable names instead of the directory names when you want to specify a directory.

You're probably already familiar with an example of this. When you log in, the shell creates the `$HOME` variable, which holds the path of your home directory. If you need to go to your home directory, you can simply type `cd $home`, which is certainly simpler to type than `cd /export/home/marco`.

In the Korn and Bourne shells, you can create an environment variable with the following syntax

```
variable=value
```

where *variable* is the name of the environment variable that you're creating and *value*

holds the path. For the C shell, the syntax is similar:

```
set variable=value
```

Suppose you maintain an ftp area for GNU utilities at `/export/ftp/pub/GNU` and a WWW server with pages stored at `/usr/local/etc/httpd`. If you find these hard to remember or type accurately, you may want to give them simple mnemonic names. Let's name them GNU and WWW for convenience.

```
$ GNU=/export/ftp/pub/GNU
$ WWW=/usr/local/etc/httpd
$ cd $GNU
$ pwd
/export/ftp/pub/GNU
$ cd $WWW
$ pwd
/usr/local/etc/httpd
```

You can save yourself a lot of time if you maintain a set of directories that you're always referring to. Please keep in mind that these environment variables will stay around only as long as you're logged in. If you want to use them all the time, you'll want to put these definitions in your `.login` or `.profile` file, as described in the article "Execute Commands When Logging In" in the April issue. ❖

