COBB

July 1997 • $11.50
VOL. 3 NO. 7

INSIDE SOLARIS™
Tips & techniques for users of SunSoft Solaris

in this issue

1
Creating a new CDE switch

3
Creating interactive network programs with shell scripts

6
Using coprocesses in the Korn shell

8
Using the typeset command to make Korn shell functions safer

9
Managing resource-intensive background processes

11
Find your files without interference

12
How does I/O redirection work?

14
Never again!

15
Put your current directory in your window title

16
And again, with zsh!

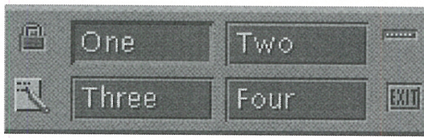
Visit our Web site at
http://www.cobb.com/sun/

Creating a new CDE switch

By W. Dean Stanton

Last month, in the article "Customizing the Workspace Switch Area," we showed you a simple customization to the Workspace switch area: We showed you how you can add, remove, and rename the buttons in the Workspace switch area, shown in Figure A.

Figure A



You can customize more than just the buttons in your Workspace switch area.

There's another customization you can make: You can define a tiny switch—like the lock or EXIT button—to appear alongside the workspace names in the Workspace switch area of the Front Panel. In fact, there's a Blank switch under the lock that has no function (other than to take up space, affecting the location of other switches).

In this article, we'll show you how to customize these switches. For our example, we'll replace the Blank switch with one that's *much* more useful: It will take us to the Compose Mail window of the mail tool!

Making our changes

In last month's article, we used the built-in features of the Common Desktop Environment (CDE) to make our changes. Customizing the switches, however, isn't quite as straightforward. To do this, we must create some new CDE setup files.

The */usr/dt/appconfig/types/C/dtwm.fp* file controls the configuration of the Front Panel. Definitions of the various elements of the Front Panel user interface widgets are located within this file. When we make our changes, we're going to create small files that override the default definitions in the *dtwm.fp* file.

You can make changes to the Front Panel for a single user by placing these override files in that user's *$HOME/.dt/types* directory. If you want to make the changes for all the users on a system, place your override files in the */etc/dt/appconfig/types/C* directory instead.

In order to prevent disruption to other users on your system, you should create the files and place them in a test account's *.dt/types* directory. Only after you've verified your changes should you place them in the */etc/dt/appconfig/types/C* directory or other users' *$HOME/.dt/types* directories.

Creating our new control

You can't add new switch controls to the Workspace switch area: It only supports four. If you want a new control, you must give up one of the others. Since the Blank control does nothing, it's easy to find a place for your first customized switch.

If you want to add another switch or so, you must decide

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
W. Dean Stanton
Production Artists ..... Marguериete Winburn
Natalie Strange
Editor ..... Karen S. Shields
Publications Coordinator ..... Linda Recktenwald
Editor-in-Chief of Online Publishing ..... Darren McGee
Circulation Manager ..... Mike Schroeder
Editorial Director ..... Linda Baughman
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

Copyright
© 1997, 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 Inc. *Inside Solaris* is a trademark of Ziff-Davis Inc. 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.

which switches you can live without. (The Busy indicator looks like a great candidate for our next switch customization, since it's marginally more useful than the Blank control.)

Delete the Blank control

Before we can create our tiny switch to start a Compose Mail window, we'll delete the Blank switch. To do this, copy the `CONTROL Blank` switch definition from the *dtwm.fp* file to *$HOME/.dt/types/Blank.fp*. Then add the `DELETE True` line (shown in blue in Listing A). You can either do so with a text editor like `vi` or by typing in the code shown in Listing A.

Please note that the switch's POSITION_HINTS value is 3. This tells the Front Panel the location of the switch.

Listing A: *Blank.fp*

```
# This is          $HOME/.dt/types/Blank.fp
# Copied from /usr/dt/appconfig/types/C/dtwm.fp
# and changed to delete the blank switch.

CONTROL Blank
{
  TYPE                blank
  CONTAINER_NAME      Switch
  CONTAINER_TYPE      SWITCH
  POSITION_HINTS      3
  ICON                FpblnkS
  HELP_TOPIC          FPOnItemSwitch
  HELP_VOLUME         FPanel
  DELETE              True
}
```

This file overrides the default definition of the Blank switch for the Front Panel.

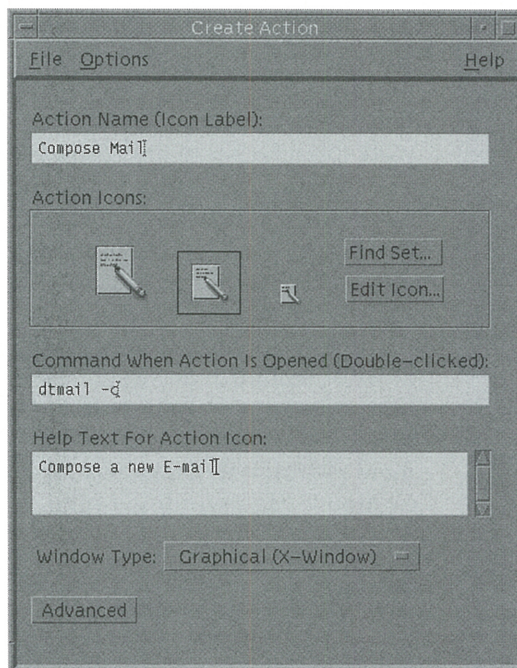
Creating a new action

Next, we'll create the action we want our new switch to use. To do so, use the `dtcreate` program to create an action named *Compose Mail*, which executes the command `dtmail -c` to open an empty compose window. When `dtcreate` makes the new action, it names it with an underscore in place of the space: `Compose_Mail`. You can't merely create the file, as it will lack the CDE checksum. You can start `dtcreate` by typing

```
# dtcreate
```

Now, just fill in the fields as shown in Figure B. To select an icon for your new action, click the Find Set... button, and browse around to find the icon you want. When you find the desired icon, write down its name: We'll be using it shortly. (If you're the artistic type, feel free to create your own icon.)

Figure B



We're creating the new Compose_Mail action.

Creating the new switch

Now we'll make our new switch control. To begin, copy the definition of `CONTROL Blank` that you just created in *Blank.fp* to *CompMail.fp*. Now modify *CompMail.fp*

to add the icon you want, PUSH_ACTION, and remove the DELETE True line, as shown in Listing B. (You may also want to update the HELP_STRING and LABEL fields while you're editing the file.)

Listing B: *CompMail.fp*

```
# This is   $HOME/.dt/types/CompMail.fp

CONTROL Compose_Mail
{
  TYPE                    icon
  CONTAINER_NAME          Switch
  CONTAINER_TYPE          SWITCH
  POSITION_HINTS          3
  ICON                    Fppenpd
  LABEL                   Compose
  PUSH_ACTION             Compose_Mail
  HELP_STRING             Compose a new E-mail
➥message using CDE's DT mail.
}
```

We've created a new control to take the place of the Blank control we just deleted.

The file *$HOME/.dt/types/CompMail.fp* redefines the switch in position number 3 to use a pen and pad icon and to execute the Compose_Mail action when you click the switch. The icon can be improved, but it's so small it barely matters.

Now that you've changed the Front Panel configuration, it's time to test it. Choose Restart Workspace Manager from the Workspace menu. When the CDE restarts, the Front Panel should show your new control. Try out the new control. If there are any problems, look for messages at the end of your *$HOME/.dt/errorlog* file to help you find the problem.

Conclusion

As you can see, the CDE is quite flexible. We've already shown you how to customize the Front Panel to suit your tastes. Since you can change the fonts, colors, and other attributes of the CDE, you can make your computer operate almost any way you want. ❖

NETWORK PROGRAMMING

Creating interactive network programs with shell scripts

By Al Alexander

Last month, in the article "Creating a Custom Network-Messaging Daemon," we began the process of creating network-aware daemons. From there, it's a short step to a client/server software application that communicates over a TCP/IP network. The unique part about our client/server communication software is that they're written in the Bourne and Korn shells, instead of a more difficult language, such as C, C++, Perl, or Tcl.

As we saw, the server program is fairly simple, and the Bourne shell is a sufficient programming language. We used telnet as our client and built a simple server that, once the client connected, would tell you how many people were logged into the machine and display some information from iostat.

This month, we're going to build on that foundation. First, we'll build an interactive server. Rather than emitting some data in a fixed pattern, our new server will service requests by the client. Then, we'll build a client that will interrogate the server.

The new server

Our new server, shown in Listing A on page 4, requires that the client identify itself, so it should work only with the appropriate client programs. Please keep in mind that this is a demonstration program and not production-quality. If we wanted to make it more secure, we'd have to add encrypted passwords and use a tool like ssh to ensure that the passwords were encrypted *before* sending them over the network.

If you connect successfully to the SysInfo server, it responds to the four requests CPU_USAGE, DISK_FREE, YOUR_NAME, and QUIT. SysInfo reports CPU usage by executing sar and displaying the results. It gets the amount of free disk space by executing df -k, running the results through grep to keep only the local file systems (i.e., those starting

with */dev/dsk*), and using `awk` to total the fourth column (the amount of free space). The YOUR_NAME request tells the server to report its host name. The QUIT command terminates the dialog, freeing the server to respond to another client.

Installing the server

Before working on the client side, let's install the server on a computer and test it with `telnet` as we did last month. To install the server, perform the following steps. (For a discussion of these steps, see last month's issue.)

Listing A: *SysInfo*

```
#!/bin/sh

# InfoServer - Demonstrate an interactive
# net daemon server.

echo "\n"
echo "Identify yourself"
read temp1
remoteHost=`echo $temp1|tr -d '\r'`

# Validate the remote computer name
case "$remoteHost"
in
   fred|FRED|joe|JOE)
      ;;
   *)
      echo "Good-bye $remoteHost"
      echo $remoteHost "attempted access on"\
         `date` >>/tmp/SysInfo.log
      exit 1;;
esac

# Process command requests
while true
do
   echo "What do you want?"
   read temp2
   theRequest=`echo $temp2|tr -d '\r'`

   case "$theRequest"
   in
      CPU_USAGE)
         sar 5 | tail -1;;

      DISK_FREE)
         df -k | grep "/dev/dsk" | awk '
            BEGIN { $total=0 }
            { $total=$total+$4 }
            END { print $total}';;

      YOUR_NAME)
         uname -n
         echo "";;

      QUIT)
         exit 0;;
   esac
done
```

This interactive network server responds to four different requests and has a rudimentary authorization scheme.

1. After you create the `SysInfo` script, give it execute permissions, and copy it to your */usr/local/inet* directory:

```
# chmod +x SysInfo
# cp SysInfo /usr/local/inet
```

2. Add the following line to your */etc/inet/services* file:

```
system_info   5001/tcp
```

3. Add the following line to your */etc/inet/inetd.conf* file:

```
system_info   stream   tcp   nowait   nobody
➥/usr/local/inet/SysInfo
```

4. Tell the `inetd` daemon to reload its configuration files:

```
# ps -ef | grep inetd
root   99     1  0 07:13:29 ?       0:00 /usr/
➥sbin/inetd -s
root  255   226  0 11:49:24 pts/0  0:00 grep
➥inetd
# kill -1 99
```

(Use the second field in the line that specifies */usr/sbin/inetd* as the parameter for the `kill` command.)

Testing the server

Once you've installed the server as a network daemon, you can test it with `telnet`. When you connect to the `SysInfo` server, it will first ask you to identify yourself. When it does so, just type *fred* or *joe*, as we did in Figure A. (As you can see from Listing A, only fred and joe can use the server.) Then, it will ask you what you want. Reply with either DISK_FREE, CPU_USAGE, or YOUR_NAME, and send it QUIT when you're finished testing. Your results should look something like Figure A.

The interactive client

The server isn't very complex. What may surprise you is that the client isn't complex either, though it *is* longer. The client program, `netClient`, is shown in Listing B. At the beginning of this program, we initiate a `telnet` session to the remote host as a coprocess (described in the accompanying article "Using Coprocesses in the Korn Shell," on page 6). Please note that we `telnet` to port 5001, which is the port where we installed our `SysInfo` server script. (Since we just tested it, we know that our server program is waiting and listening on this port on the remote workstation.)

The main body of the client first identifies itself by waiting for the server (via the coprocess) to request the client's name, upon which the client responds *FRED*. We describe the `WaitFor`, `Say`, and `GetReply` functions in the coprocess article. The client uses the same procedure to get the amount of free disk space and CPU usage for the server, then terminates the session by issuing the `QUIT` command.

Notes

It's important to pay careful attention to synchronization in your client and server scripts. If both the client and the server are reading, your program will hang because the programs are waiting on each other. However, fear not, because this is always a logic problem that you must deal with when creating this type of application—just make sure your logic is right.

Figure A

```
# telnet widget2 5001
Trying 140.244.96.203...
Connected to widget2.
Escape character is '^]'.
Identify yourself
fred
What do you want?
CPU_USAGE
14:54:14        0       1       0      99
What do you want?
DISK_FREE
224243
What do you want?
YOUR_NAME
widget2
What do you want?
QUIT
Connection closed by foreign host
```

Here we use `telnet` to test by hand all the commands offered by the `SysInfo` server.

Listing B: *netClient*

```
#!/bin/ksh
# netClient - read system information from the
# specified remote computer

#########################
# FUNCTION DEFINITIONS #
#########################

usage()
   {
   echo "usage: netClient <host>"
   echo "Display CPU usage and free disk space"
   echo "available on <host>"
   exit 1
   }

# Send a string to the coprocess
Say()
   { print -p "$1"; }

# read a string from the coprocess
GetReply()
   {
[illegible]
          wer
          ver

          lines from coprocess until we find
         one we're waiting for...
waitFor()
   {
   typeset stringToWaitFor temp input
   stringToWaitFor="$1"
   while true; do
      read -p temp
      input=`echo $temp|tr -d -c '? [A-z]'`
      if [ "$input" = "$stringToWaitFor" ];
then
         return
      fi
   done
   }

################
# MAIN PROGRAM #
################

# Ensure that we have the right # of arguments
if [ $# != 1 ]; then usage; fi

# Establish a connection to the remote computer
telnet $1 5001 |&

# Wait until we get the prompt from the remote
# computer, and reply
WaitFor  "Identify yourself"
Say      "FRED"

# Wait until remote computer asks what we want,
# then tell it we want the DISK_FREE information,
# then read the information
WaitFor  "What do you want?"
Say      "DISK_FREE"
diskFree="`GetReply`"

# Now use same procedure to get CPU USAGE
WaitFor  "What do you want?"
Say      "CPU_USAGE"
cpuUsage="`GetReply`"

# Terminate the session to free the remote
WaitFor  "What do you want?"
Say      "QUIT"

# Display the results
echo "DISK_FREE:     $diskFree"
echo "CPU_USAGE:     $cpuUsage"
```

Our network client program can ask for specific information from our server using a simple dialog.

Conclusion

In this article, we've demonstrated the steps necessary to create a client/server communication program that you can customize to meet the demands of your own environment. We wrote our programs using only Bourne and Korn shell scripts, so any administrator familiar with shell scripts can create customized network-ready applications.

Our program is based on a conversational approach between the client and server, which you can easily extend to meet your needs. This conversational approach is very flexible: Not only is it easy to debug with `telnet`, but with care and planning, you can write a server that can communicate with different types of clients. ❖

SHELL PROGRAMMING

Using coprocesses in the Korn shell

By Al Alexander and Marco C. Mason

Do you like to write shell scripts? Would you really like to change the way certain applications run? If you answered yes to both of these questions, then you owe it to yourself to investigate a great feature of the Korn shell: coprocesses. Using coprocesses, you can write a new front end for an application and make it look and operate any way you want, without rewriting it. In this article, we'll show you how.

What's a coprocess, anyway?

OK, I'm guessing that the question running through your mind about now is "What the heck is a coprocess?" Before I answer that, let's take a brief look at how we execute programs inside shell scripts. Listing A shows a trivial shell script that counts the number of files in your directory.

Listing A: *numFiles*

```
#!/bin/ksh

# numFiles - display the number of files in
# the specified directory

echo "There are \c"
NF=`ls -al $* | grep ^- | wc -l`
echo $NF "files. "
```

Execution of this shell script is suspended until the commands in blue finish.

The Korn shell, when executing this shell script, processes it line by line until it finds the line in blue. The shell then must execute a pipeline of commands, and it does so, suspending operation of the current script. When the line in blue finally completes, the shell resumes execution of the `numFiles` script.

The current script stops executing while the other processes run. Your shell script can't do anything useful while processing the `ls`, `grep`, and `wc` commands. In this case, that's no tragedy, since we don't expect `numFiles` to take an appreciable amount of time to execute.

But what happens when the command pipeline process takes a long time to run? The script appears to be hung—most people worry, at least a little bit, when the computer doesn't appear to be doing anything.

To get around this annoyance, you could run the procedure as a background task. This way, your script can execute while the background tasks work away. But how do your processes communicate?

This is where coprocesses come in. A coprocess runs in the background but opens a special pipe that your application can use to read and write to. Thus, if your scr[illegible] forming some time-consuming [illegible] ground, your script can perform o[illegible] such as informing the user of the pr[illegible] the operation, while the coprocess is w[illegible] If you want to get fancy, you can even g[illegible] your users an estimate on how long the jo[illegible] should take and how much progress it's made.

Having said all that, we offer a simple definition for a coprocess: A coprocess is a process that can run concurrently with your main process. Neither process waits for the other to complete; instead they can communicate with each other and cooperate to perform a task.

Creating a coprocess

Now that we know what a process is for, we need to know how to create and interact with one. We can do this by ending a command with the operator `|&`. When you look at how it works and what it does, the operator even makes sense. This operator does two things: First, it creates a pipe that your main process can use to communicate with it, just as if you used the piping operator `|`. Second, it executes the specified command in the background, just as if you used the background execution operator `&`. The `|&` operator to start a coprocess is a logical extension of these. An example of starting a coprocess might look like this:

```
# ps -ef |&
```

This command runs the `ps -ef` command as an asynchronous background process, just as if you'd used the `&` symbol alone. However, instead of having the output come directly to your screen, you can now read it from a two-way pipe at your convenience. To read a line of data from the coprocess, you simply use the `read -p` command, like this:

```
# read -p line1
```

This command reads the first line of input from the coprocess and stores the contents of the input in the shell variable named `line1`. What does this variable contain? You can display it with the `echo` command:

```
# echo $line1
UID   PID  PPID  C    STIME TTY       TIME CMD
```

As you can see, the variable `line1` contains the first line of output from the `ps -ef` command. If we perform these `read` and `echo` statements again, we can obtain the next line of output from the `ps -ef` command:

```
# read -p line2
# echo $line2
    root    0     0  0 08:43:35 ?     0:01 sched
```

You can send a line of data to a coprocess with the `print -p` command. In this particular example, there's no need to write any information to the asynchronous process, because the `ps -ef` command won't listen to what we write.

However, some Solaris commands are interactive and will accept input. For example, you might want to do some extensive calculations in your shell script, so you could fire off the `dc` (desktop calculator) program as a coprocess. Then your script can get the calculator to perform all the calculations.

Making it more readable

In order to make coprocesses even easier to use and the shell scripts easier to read, we've created three shell functions to communicate with a coprocess: `Say`, `GetReply`, and `WaitFor`, as shown in Listing B. Because it's the simplest of the three, let's look at `Say` first.

Listing B

```
Say()
   { print -p "$1"; }

GetReply()
   {
   typeset answer
   read -p answer
   echo $answer
   }

WaitFor()
   {
   typeset stringToWaitFor temp input
   stringToWaitFor="$1"
   while true; do
      read -p temp
      input=`echo $temp|tr -d -c '? [A-z]'`
      if [ "$input" = "$stringToWaitFor" ]; then
         return
      fi
   done
   }
```

These three functions make it easy to communicate with a coprocess.

The `Say` function does only one thing: It takes whatever argument you give it and sends that argument to the standard input stream of the coprocess. So if we execute the command

```
Say "DISK_FREE"
```

the `Say` function takes the string DISK_FREE and writes it to the standard input of the coprocess.

The `GetReply` function is almost as simple as the `Say` function. It does the opposite of the `Say` function, reading a line of output from the coprocess with the `read -p` command. It stores the server's reply in the variable `answer`, then echoes this `answer` to the main process' standard output stream. (To make the function more portable, `GetReply` makes the variable `answer` a local variable by using the `typeset` command. See the accompanying article "Using the Typeset Command to Make Korn Shell Functions Safer" on page 8).

The `WaitFor` command, like `GetReply`, uses the `read -p` command to read from the coprocess. In the `WaitFor` function, the line

following the `read -p` command uses the `tr` command to delete all characters but letters, spaces, and the question mark. Then we copy the result to the variable `input`. If the value in input matches the string we're looking for, then `WaitFor` returns—otherwise, it continues to read lines until it finds a match.

We removed some characters from the input string to simplify pattern matches. For example, if a command issues a number as part of its response and you don't know what number it will use, you can still use `WaitFor` because it removes the digits. It also removes extraneous carriage returns from the input.

Using a coprocess

With these handy shell functions, you can easily work with coprocesses. For example, the client program in the article "Creating Interactive Network Programs with Shell Scripts" on page 3 uses `telnet` as a coprocess to manage communications over the network. The part of the client that controls the dialog between the `telnet` coprocess and the main process looks like this:

```
# Wait until we get the prompt from the remote
# computer, and reply
WaitFor  "Identify yourself"
Say      "FRED"

# Wait until remote computer asks what we want,
# then tell it we want the DISK_FREE
# information,then read the information
WaitFor  "What do you want?"
Say      "DISK_FREE"
diskFree="`GetReply`"
```

The shell functions make it very clear what's expected. Using our customized chat language, we wait for the server to ask us who we are with the *Identify yourself* prompt. When the client receives this prompt, it responds by "saying" that we are FRED. The server then asks what we want, and we request the DISK_USAGE for the host. We then store the server's reply in the `diskFree` variable for future use.

Synchronization

You should be aware of synchronization issues when using coprocesses. If you execute a `read -p` command to read information from a coprocess, your main process must wait until the information becomes available. Similarly, if the coprocess needs input before it can act, it will wait for the input. If you're not careful, you can achieve a deadlock in which each process is waiting for the other. As always, you must carefully design your shell scripts to be tolerant of errors.

Conclusion

Now you're ready to use coprocesses in your own shell scripts. You can, if you like, build a front end for an editor, other shell scripts, etc. Anytime you have an interactive application or an application that simply takes a long time to execute, you can implement it as a coprocess to let the user retain control of the terminal and still allow your application to control the background task. ❖

SCRIPT TIP

Using the typeset command to make Korn shell functions safer

By Al Alexander

If you're in the business of writing Korn shell functions that you want to make portable, then you need the `typeset` command. The `typeset` command, used inside a Korn shell function, makes your function variables local in scope, so you don't accidentally overwrite global variables created in other portions of a Korn shell script.

Discussion

In the Bourne shell and the Korn shell, the normal process of creating a variable makes that variable globally available throughout your entire shell program. As an example, if you're in a function and you write

```
USER_NAME="fred"
```

the variable `USER_NAME` is created as a global

variable. This means that the contents of the variable USER_NAME can be changed not only in your function, but anywhere else in the program. Even worse, if the variable USER_NAME existed before your USER_NAME="fred" statement, you just overwrote the value previously stored in USER_NAME! This has long been a problem with global variables (in any language, not just the Bourne and Korn shells).

The solution to this problem, available in the Korn shell, is to use the typeset command to make USER_NAME a local variable in your function. In this way, your function can use a variable named USER_NAME and manipulate its value without clobbering a variable used elsewhere in the program that happens to use the same variable name. If you're trying to create reusable functions, the typeset command will save you many headaches.

Listing A shows a sample Korn shell application, named typeset.sh, that demonstrates the difference between local and global variables. If you run this program on your workstation, the result will be

```
A:      A was created in MAIN ...
B:      B was changed in Func ...
```

Both variable names A and B are used in the main program and in the function Func. Because the typeset command makes variable A local to the function, the value of the A variable used in the main program isn't clobbered when the main program calls Func, while the value of B is destroyed by the function.

Listing A: *typeset.sh*

```
#!/bin/ksh

function Func
   {
   typeset A
   A="A was changed in Func ..."
   B="B was changed in Func ..."
   }

#---------((  MAIN  ))-------#

# Set up our initial values
A="A was created in MAIN ..."
B="B was created in MAIN ..."

#  call the sample typeset function
Func

# print the results
echo "A:         $A"
echo "B:         $B"
```

This program demonstrates the difference between local and global variables.

The typeset command actually has many features for specifying how a variable acts. But perhaps its most important behavior is that it allows you to create a local variable inside a function. Be sure to read the man page for typeset to see what other uses it may have in your shell scripts. ❖

PERFORMANCE TIP

Managing resource-intensive background processes

Multitasking allows you to squeeze every bit of utility from your computer. If you're simply editing a file at your workstation, you're not asking much of your computer (unless you're using emacs). Solaris provides a multitasking environment so you can run multiple programs at once and accomplish your work as quickly as possible. While you're typing away at your E-mail message, your computer could be analyzing map data, sorting files, running SPICE, etc.

Sometimes, however, running a large job in the background makes the system sluggish enough to be counterproductive. When the system response is too slow, it prevents you from working effectively. If the background job must be done now, you have to live with it. On the other hand, if you can postpone the

job, you can try a couple of things to improve system performance.

Don't waste the work!

Obviously, one way to speed up the system is to kill the offending process. For example, if process 1206 is slowing down your system, you can kill it like this:

```
# kill -KILL 1206
```

However, if that process has been plugging away for hours, killing it wastes all the work already done. Once you restart the process, it'll have to do all that work again.

Make the process play nicely

A better solution is to tell Solaris to give less time to the process that's slowing down the system. The **renice** command, shown here, allows you to tell Solaris the relative priorities of processes on your computer:

```
# renice 19 1206
```

Here we just told Solaris to run process 1206 at a priority of 19. The priority levels range from -20 (the highest) to 19 (the lowest). Thus, the **renice** command we just used told Solaris to give process 1206 as little CPU time as possible.

If you're a privileged user, you may specify any priority from -20 to 19, while normal users may only specify values from 0 to 19. Although any user may slow down his or her own processes, few do. You might want to educate any users on the system that system performance will be snappier if they reduce the priority of any processes running in the background. However, often it's up to you to lower the priority of any processes that cause the system to be sluggish.

Just get it over with

Sometimes, if the job is important or time-critical, it's better to just get it over with. You can either continue to work while the system is sluggish, or you can bite the bullet and finish the job as quickly as possible. To do so, you could **renice** the process to a higher priority. Sure, the system will become even more sluggish, but the sooner it quits, the sooner everything gets back to normal.

Alternatively, you could stop any unnecessary tasks, raise the offending process' priority, and advise everyone to take a long lunch. Once the process finishes, you can restart the other tasks, and everything should return to normal.

Be careful when you increase the priority of the task. If you increase the priority too much, you can make the system unusable for everyone.

Stopping and restarting

Here's a handy trick for your toolbox: You can stop and start processes with the SIGSTOP and SIGCONT signals. So if the process that's making your system sluggish can wait until the evening to run, you can just stop it and start it again before you leave for the day. To stop the process, you use the **kill** command, like this:

```
# kill -STOP 1206
```

Later, when you want the process to continue, just restart it like this:

```
# kill -CONT 1206
```

This technique allows you to keep the work that the process has already done. Since the process is stopped, as other processes need RAM, the process will be swapped to disk. This is the major drawback to this technique: You're tying up swap space that other processes could use. As long as you don't run out of swap space, it's not a problem. However, when you run out, Solaris will start killing processes. Unfortunately, it may kill the one you're trying to defer.

Please note that you *must* remember to restart the process, or the process' owner will have to wait even longer for the results. Rather than trust your memory to restart the process, use the **at** command to tell Solaris to restart the job. You can do so like this:

```
# at 1800
at> kill -CONT 1206
at> [Ctrl]D <EOF>
warning: commands will be executed with /bin/sh
job 861400800a.a at Fri Apr 18 18:00:00 1997
```

Now you can rest assured that the job will restart, unless Solaris kills it because the system ran out of swap space. Fortunately, the **at** command sends you E-mail with the results of your command. Be sure to check your mail before you leave, so you can see if the process restarted successfully. If not, you must resubmit the job or tell the process' owner to do so.

Which jobs are slowing you down?

Of course, in order to manage processes that are slowing down your system, you must identify them. Quite often, you'll already know which ones they are. If you have a good suspicion as to which process is currently bogging you down, you can help confirm it by using the `ps -ef` command to check the time that the process started.

The easiest way to identify the jobs that are slowing down your system is to use the `top` command. This command doesn't come with Solaris, but you can find the source code, as well as precompiled versions, from many sources on the Internet.

Normally, you'll find that the jobs that slow you down are memory-intensive rather than CPU-intensive. Most slowdowns that we've experienced are related to the amount of disk I/O the process causes.

Summary

While it's true that multitasking allows you to get the most work out of your computer per unit time, sometimes the performance degradation of some processes actually lowers your productivity. In cases like this, you can use the `renice` command to tell Solaris to not work so hard on a particular process. If that doesn't work, you can stop the job during the peak load, then restart it later at a more convenient time. ❖

SYSTEM ADMINISTRATION

Find your files without interference

Have you ever tried to use `find` to locate a specific file? Of course you have! Since `find` can take a long time—especially on a large network with lots of disk space—you probably opened a terminal window while `find` was running and got to work on another part of your project.

You may not have found the file on your first attempt. Unless you have permission to enter every directory on all the file systems, your terminal buffer may have been filled with error messages telling you where the `find` command couldn't look, like this:

```
# find / -name hello.c -print
find: cannot read dir /lost+found: Permission
➥denied
/tmp/hello.c
find: cannot read dir /usr/lost+found:
➥Permission denied
   •  •  •
```

The file you were looking for may have scrolled off the top of the screen, perhaps far enough to be unrecoverable through the terminal's scroll-back buffer, if it has one. (In our test, after finding */tmp/hello.c*, `find` printed 58 error messages.)

Getting rid of the error messages

Why does the `find` command tell you it can't examine specific directories? It does so to inform you that even if it doesn't find the file you're searching for, the file may exist somewhere on the system. For many purposes, however, if you can't locate the file in directories where you have permission, then it may as well not exist. In cases like these, you may not want all those error messages on your screen.

How can we prevent all these error messages, so we can see the information we want? Since the `find` command sends these error messages to the error stream, one way to do so is to redirect the standard error stream to */dev/null* so that the error messages are discarded. This leaves you with just the information you want: the location of your file, if it exists.

When you read the article "How Does I/O Redirection Work?" on page 12, you'll see that you can discard the error output for the Bourne and Korn shells, leaving only the information you want, like this:

```
# find / -name hello.c -print 2>/dev/null
/tmp/hello.c
```

You can also separate the standard output and standard error streams in the C shell, albeit not as nicely.

Separating the wheat from the chaff

However, we can take another approach—one that works in *all* shells. What do we want? We want the output to the program. We don't

really care about discarding the error messages, we just don't want them in our way, and we don't want them causing the desired information to scroll out of the terminal buffer.

For this trick, we simply let the error messages print to the terminal, and we'll print the program output at the end, *after* all the error messages. This way, all the information we want is easy to access, and we don't have to worry about an excessive quantity of error messages scrolling the data off the screen.

We can do this by piping the standard output stream of the `find` command (or other command) through one of the filter commands, like `cat` or `more`. (We typically use the `more` filter, and we suggest that you do so as well, because if the command generates more than one screenful of data, you'll probably want to page through it anyway.) Thus, our `find` command now looks like this:

```
# find / -name hello.c -print | more
find: cannot read dir /lost+found: Permission denied
find: cannot read dir /usr/lost+found: Permission denied
    •   •   •
/tmp/hello.c
```

This works because both the `find` command and the `more` command want access to your terminal. Since `find` is first in the pipeline, it gets the terminal first. The `more` command, meanwhile, is collecting the output of the `find` command. Only after the `find` command ends and releases the terminal does `more` get to use the terminal to display its data, the data you wanted, after all the error messages.

Wrapping it up

If you're always running from terminal to terminal helping your users, you'll probably want to use the latter trick. This way, you won't have to see what shell the user is running to know the syntax for redirecting the standard error stream. However, if you're always at your own terminal, redirecting the standard error stream is sometimes the fastest way to get the job done, because displaying to the terminal can be a slow process. If the command you're executing generates a huge amount of error text compared with a small amount of data, all the time spent writing to the terminal is wasted. ❖

BASIC CONCEPTS

How does I/O redirection work?

If you've been using Solaris, or any other version of UNIX, for any length of time, you've encountered I/O redirection in one form or another. But have you wondered how it works? In this article, we'll show you the underpinnings of I/O redirection and piping, so you can take advantage of these features to build pipelines to perform complex tasks.

Standard streams

The concept that UNIX commands follow to maximize their flexibility is that of *standard streams*. Whenever you run a command, UNIX provides three standard I/O streams to the command: The command reads any input it needs from the *standard input* stream, writes its results to the *standard output* stream, and writes any error messages to the *standard error* stream.

One of the greatest features of UNIX is that the command you execute doesn't set up these streams, rather, the program that starts the command does. (Usually, this is the shell you're using.) Since the calling program, rather than the command, sets up the standard I/O streams, every command must agree on the order and meaning of each stream.

Normally, when you execute a command from your shell, the shell opens an input stream from your terminal and two output streams to your terminal. These streams are passed to the command as its standard input, output, and error streams.

If the program reads input, by convention it's supposed to read it from the standard input stream. Similarly, if it provides output (and what command doesn't?), it writes it to the standard output stream, and if it encounters any errors, it's supposed to write them to the standard error stream.

So in typical use, when you type a command at the console, like this

```
# ls
hello.c     a.out               makefile
```

the ls command doesn't know which terminal to write its results to. It simply writes them to the stream provided by your shell.

If your shell wanted to, it could send the output of the ls command to a different location. It can do so without the ls command knowing about it. In fact, your shell has two special characters, < and >, that tell it to override the normal settings for the standard input and standard output streams.

The < symbol tells the shell to use the next word as the name of a file to open and use as the input stream. Similarly, the > symbol tells the shell to use the next word as the name of a file to open and use as the output stream. For example, the command line

```
# ls <hello.c >goodbye
```

tells the shell to open the file named *hello.c* and use it as the standard input stream, open the file named *goodbye* and use it as the standard output stream, and run the ls command with these streams. (Please note: Since we didn't override the definition of the error stream, any errors will still appear on your terminal screen.) The order doesn't matter. This command line has the exact same result:

```
# ls >goodbye <hello.c
```

If the input file doesn't exist, your shell will issue an error telling you so. If the output file doesn't exist, the shell will automatically create it.

The cat command

The cat command is probably the simplest command in Solaris. It simply reads a line from the standard input stream and writes that same line to the standard output stream. It continues to do this until it reaches the end of the standard input stream. Here, we run the cat command and type two lines:

```
# cat
now is the time for all good men to come
now is the time for all good men to come
to the aid of their party.
to the aid of their party.
```

Then we tell cat to stop by sending the end-of-file command (normally [Ctrl]D). By redirecting the standard input of the cat command, we can type the contents of a file, like this:

```
# cat <hello.c
#include <stdio.h>
int main(int, char **)
    { printf("Hello, world!\n"); }
```

Similarly, we can create a file by redirecting the output of cat, like this:

```
# cat >junk
now is the time for all good men to come
to the aid of their party.
```

We can even copy files using cat, just by redirecting both the input and output streams:

```
# cat <hello.c >goodbye.c
```

Command pipelines

Now let's move on to command pipelines. Often, you can send the output of one command through another command to perform a more complex task. Suppose, for example, that you wanted to know the number of processes running on your computer. Using I/O redirection, you can use the following two statements to do so:

```
# ps -ef >tempFile
# wc -l <tempFile
41
```

Here, we use the ps command to create a list of all the processes, one per line, and put them in *tempFile*. Next, we count the number of lines in *tempFile* with the wc command to get our result, 41.

This method is a bit clunky. We get the results we want, but we now have a useless file that we need to remove, *tempFile*, cluttering up our directory. If we're going to string more commands together, we must create more temporary files. Even worse, if we happen to be in a directory where we don't have write privileges, the shell will complain to us that it can't create *tempFile*.

The shell provides a much better way to string commands together to form a pipeline. We can use a vertical bar to separate the commands and omit the other I/O redirection clauses, like this:

```
# ps -ef | wc -l
```

The shell breaks the command line into two pieces at the vertical bar. Then it opens a special temporary file as the standard output of the ps command and opens that same file as input for the wc command. It gives you

basically the same thing, but it automatically manages the temporary files for you! The special temporary files aren't stored in your current directory, so even if you have no write privileges in the current directory, you can pipe commands together with the vertical bar.

One nifty thing about a pipeline is that Solaris runs the commands at the same time. It doesn't wait for the `ps` command to complete before starting `wc`; it starts them both at once. As the `ps` command outputs each line, the `wc` command can read it. This helps Solaris keep the temporary file size small because after `wc` reads a line, that line no longer needs to be stored. That's another advantage of pipelining—Solaris can keep the temporary file in memory, rather than writing it to disk, so the pipeline may execute faster. (We say *may* rather than *will* because if you do it the hard way, and your temporary file is short, the temporary file may be buffered in RAM anyway.)

Redirecting the standard error stream

Earlier, we discussed the three streams but only described how to redirect standard input and output. We put off the discussion of redirecting the standard error stream because it doesn't operate the same way for all the shells. If you're using the Bourne or Korn shell, you can redirect the standard error stream by using the 2> operator.

```
# find / -name hello.c -print 2>/dev/null
```

The 2> tells the Korn shell that you want to redirect stream 2 (the standard error stream), and */dev/null* is where you want it to go—the bit bucket, in this case.

However, if your favorite shell happens to be the C shell, you're in for a disappointment: The C shell uses the `>&` operator to redirect the standard error *and* output streams to the specified location. It doesn't allow you to redirect the standard error stream independently of the standard output stream. You can, however, trick the C shell into doing some of what you want, like this:

```
Ringo% (find / -name hello.c >/tmp/x) >&/dev/null
```

Here, we execute the `find` command within a parenthesis. This tells the shell to run everything inside the parenthesis as a single command. So the shell redirects the output of the `find` command to a temporary file. Then outside the parenthesis, the C shell redirects whatever output is left to the bit bucket. The shortcomings of this are that you don't get the desired output on the console and you're creating a temporary file. So, to see the results, you must display the temporary file, then delete it:

```
Ringo% cat x
/export/home/marco/hello.c
Ringo% rm /tmp/x
```

Summary

I/O redirection and piping are powerful concepts. I/O redirection allows you to change the behavior of your programs and shell scripts without having to modify them. Piping builds on the base of I/O redirection to give you the ability to string chains of commands together to perform useful tasks.

For even more information about piping and I/O redirection, be sure to read the `man` page for sh, ksh, and csh. ❖

LETTERS

Never again!

Our response to Jonathan Gripshover's question in the April issue was clearly incorrect. After searching the Web, reading Sun's documentation, and even asking an engineer at Sun, we came to the conclusion that you can't change the icon and title text for an `xterm` window. Several readers not only came forward with the escape codes that Jonathan wanted, but they provided references, sample code, and other useful tips as well!

Scott Gorton pointed out that you can find a table of `xterm` escape sequences in Appendix E of O'Reilly and Associates' *X Windows System User's Guide* (Motif edition).

Gary Andresen also responded. We said that there are no escape codes for `xterm` windows, but he knows there are from his days as an X11 programmer back in the X11R3 and X11R4 days. (See the article on page 15 based on his letter.)

Table A shows the escape codes you can use to set the window's icon name and title to the specified text. Please note that ^[(in blue) represent the [Esc] character, and ^G (also in blue) represents the [Ctrl]G character.

So we can change a window's title to the string *Hello* by typing the following command:

```
# echo "^]0;Hello^G"
```

(This will work in the Korn, Bourne, and C shells. In the bash shell, however, you must type [Ctrl]V before the [Esc] and ^G characters. Other shells may behave differently.)

Table A

^[ ] 0 ; *text* ^G	Set window's icon name and title to *text*.
^[ ] 1 ; *text* ^G	Set window's icon name to *text*.
^[ ] 2 ; text ^G	Set window's title to *text*.

These are the Set Text Parameters escape codes for the `xterm` window.

Some readers may recognize these escape sequences: They're the same ones that `dtterm` uses. It turns out that not only do the escape sequences exist, but they're ones we're already familiar with! ❖

Put your current directory in your window title

This is the rest of Gary's letter that we referred to in the previous section.

One of the ways I use the title bar of the `xterm` window is to print a simple prompt at the command line and print the full working directory on the title bar. This way, if you're 10 levels deep in a directory structure, your prompt is just 10 characters or so (your mileage may vary), but you can read the entire working directory string on the title bar!

Here [in Listing A] is part of my *.profile* file for `ksh` to set up the window title so it changes whenever I change directories.

Gary Andresen
Andresen Consulting Services
gary.andresen@qiclab.scn.rain.com

Listing A: *section of .profile*

```
# Additions to .profile to change the xterm
# title to the current working directory

# Shell function to set the window title
label () { print -n ${wTTLpre}$*${wTTLpost}; }

# Alias to set my prompt
alias cdprompt='PROMPT="$USER> "; export PROMPT; PS1=$PROMPT'

# Set the window title to user, host and directory
alias stripe='label $USER $HOSTNAME - $PWD'

# Function to change directory & update title
cds () { "cd" $*; eval stripe; eval cdprompt; }

# Alias: use 'cds' function when you type 'cd'
alias cd=cds

# Set up initial terminal environment
case $TERM
in
   xterm|dtterm)
      # For xterm or dtterm, put path in title
      wTTLpre="\033]2;"
      wTTLpost="\007"
      ;;
   sun-cmd)
      # Do the same for a sun-cmd window
      wTTLpre="\033]l"
      wTTLpost="\033\\"
      ;;
   *)
      # For other terminals, just print the path
      wTTLpre="\012"
      wTTLpost="\012"
      ;;
esac

eval stripe
PS1=$PROMPT
export TERM
stty sane
```

Gary's use of the window's title bar demonstrates several great tricks above and beyond keeping the current path available and out of the way. First, Gary's code had to take over the `cd` command, so you can change the window title when you change directories. He solved this problem by creating a function called `cds` that executes the `cd` command with the parameters passed to it, then evaluates the `stripe` and `cdprompt` aliases, which update the window title and prompt, accordingly:

```
# Function to change directory & update title
cds () { "cd" $*; eval stripe; eval cdprompt; }
```

However, most of us are accustomed to typing `cd` rather than `cds` to change to a different

PERIODICALS MAIL

SunSoft Technical Support
(800) 786-7638

Please include account number from label with any correspondence.

directory. If Gary left it at that, the trick would be interesting but unreliable: Whenever someone accidentally used `cd` instead of `cds`, the window title would be wrong.

Gary overcame that problem with another trick: He created an alias for the `cd` command that invokes the `cds` alias:

```
# Alias to use 'cds' function when you type 'cd'
alias cd=cds
```

Isn't that a problem? If your alias calls the `cds` function, which executes `cd`, which calls the `cds` function, etc., how does it ever stop? In order to prevent an infinite loop, he told the `cds` function not to use the alias, by quoting the `cd` command! Thus, when you type `cd`, the alias for `cd` tells the shell to execute the `cds` function, which then executes the *real* `cd` command. Then the `cds` function updates the command prompt and window title and exits.

And again, with zsh!

Another reader also responded with a similar trick. However, rather than use the Korn shell, James Clough uses `zsh`*. Thus, he gets the same result by going in a different direction.*

On the back page of the April 1997 issue of *Inside Solaris*, you conclude that `xterm` "supports no escape sequences outside of the VT102 and Tektronix set." Unfortunately, your search didn't extend to the "Xterm Control Sequences" document in the *xc/doc/specs/xterm* directory of the X11 distribution, available from *ftp.x.org* and its many mirrors.

Personally, I make use of `xterm`'s title bar with the following `zsh` functions:

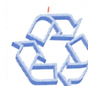
```
function title
   {
   # Put path or parameter string on the title bar
   if [ "$#" -gt 0 ]; then
      newTitle='echo $*'
   else
      newTitle="'hostname':'echo $PWD'"
   fi
   case $TERM in
      xterm*|dtterm*)
         # Change title of terminal window
         echo "^[]2;$newTitle^["
         ;;
      *)
         # Some terminals don't have a title bar
         ;;
   esac
   unset newTitle strLen
   }

chpwd()
   {
   [[ $PWD = $HOME ]] || ls -AFC|head -6;title
   }
```

The `zsh` shell automatically calls function `chpwd` every time you `cd`, `pushd`, or otherwise change directories. The `chpwd` function, above, lists the first few files in that directory, then calls `title` to update the window. You may also call the `title` function with arguments to display those on the title bar.

While I'm writing to you, anyone who read "A Simple Way to Find Files Faster," also in the April issue, may be interested in the GNU `locate` utility (in the GNU `findutils` package, which you can find on *prep.ai.mit.edu* and mirrors). It provides an `updatedb` program to run as a `cron` job and `locate` to search the resulting database.

James Clough
clough@mpdserver.ntc.nokia.COM

Here, James uses the title bar in much the same way Gary does. However, he's using a different shell: `zsh`. One feature of this shell is that it automatically executes the `chpwd` function whenever you change the path. So all James had to do was decide what he wanted. In his case, he wants to display the first six lines of the directory he changes to, unless it's his home directory. He also changes the window title to display the current path. There's more than one way to skin a cat! ❖



Printed in the USA
This journal is printed on recyclable paper.