May 1998 • $11.50
Vol. 4 No. 5

ZIFF-DAVIS
THE COBB GROUP

INSIDE SOLARIS™

Tips & techniques for users of SunSoft Solaris

IN THIS ISSUE

1
Make a shell script mail you a summary

3
An introduction to awk

7
PERL of the Internet

9
Multithreaded programming in UNIX

11
Changing the title bar of an xterm window

13
Turning the File Manager into your command center

Visit our Web site at

Make a shell script mail you a summary

by Marco C. Mason

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 it accepts a mail message via standard input. Once the command 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 own 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}
```

ZIFF-DAVIS a SOFTBANK company

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
Local	(502) 493-3300
Customer Relations fax	(502) 491-8050
Editorial Department fax	(502) 491-4200

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	John Gowin
Contributing Editors	Al Alexander
	Marco C. Mason
	August Mohr
	Brian Schaffner
Print Designer	Marguerite Winburn
Editors	Karen S. Shields
	Joan McKim
	Michael E. Jones
Publications Coordinator	Linda Recktenwald
Product Group Manager	Michael Stephens
Circulation Manager	Mike Schroeder
Publisher	Jon Pyles

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.

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

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

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 consists merely 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 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.

Normally 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 create the file 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 drawbacks: 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 with the lines

```
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.

With this technique you just 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 in either of two ways:

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

or

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

Capturing error output

So far, we've shown you only 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 must 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}
```

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 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 a time-saver. It's nice to be able to run your scripts and programs knowing that you can examine the results at your leisure rather than watching for the output before it scrolls off the screen. Another 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. ❖

UNIX POWER TOOLS

An introduction to awk

by August Mohr and Marco C. Mason

The `awk` language is a powerful part of the UNIX system. This language allows you to do many things that would otherwise require you to write a C program. In fact, it's often used for prototyping programs to test ideas before converting them to C or another compiled language. If you're wondering about the funky name, it comes from the initials of its authors: Aho, Weinberger, and Kernighan.

If you've never used `awk`, this article may inspire you to try it. If you've used only its default output capabilities, we'll give a starting point for taking fuller advantage of this powerful tool. Be sure to read the `man` page for more information on `awk`'s capabilities.

Despite the power, `awk` programs can be amazingly simple—especially when compared with the equivalent C programs. This simplicity comes in part from `awk`'s ability to make assumptions about the format of its input that a generic programming language can't. `awk` presumes that its input is ASCII text, that the input can be organized into lines or records, and that the records can be organized into fields.

Simple idioms, complex tasks

The simplicity possible with awk allows you to use it within pipelines for such tasks as extracting fields from a line of input. Here's an example of a common idiom used in shell scripts. By default, awk will use a blank space in an input line (spaces and tabs) to separate fields of non-blank characters. The following awk command will print the first field of every line in the file *test*:

```
$ awk '{ print $1 }' test
```

The quoted part of the command line tells awk what to do with the file *test*. In this case, it tells awk to print the first field in each line.

The command we just showed you illustrates the basic structure of an awk command. The part in quotes specifies a list of action statements telling awk what to do, and the part after the quotes specifies the list of filenames to process.

Records and fields

When awk processes a file, it reads it line by line. Each line is considered to be a record and is treated by itself. Each record is a collection of fields, separated by white space.

In your action statements, you can refer to a field by the construct $*n*, where *n* is the field number you're interested in. In an awk program, you may have multiple action statements, each in the form

```
pattern { action }
```

where you can omit either the pattern or the action part. If you omit the pattern part, awk executes the specified action on every line. In our example, we omitted the pattern part and used just the action part, which simply prints the first field in the record.

Let's try it out. First, go to your root directory, type ls -C, and examine your output:

```
$ ls -C
TT_DB   dev       export   lib   opt        sbin    usr   xfn
bin     devices   home     mnt   platform   shlib   var   xxx
cdrom   etc       kernel   net   proc       tmp     vol
```

Now, if we pipe the output of the ls -C command to our example awk command, we should see the following result:

```
$ ls -C | awk '{ print $1 }'
TT_DB
bin
cdrom
```

For each line, awk reads the record, splits it into fields, and prints the first field.

Pattern matching

The pattern part of an action statement tells awk when to execute the action. If the pattern doesn't match, then awk doesn't execute the corresponding action in curly braces.

If you want to operate only on lines that contain a particular string, you can use the syntax /*x*/ to represent the string, where *x* is the string you're looking for. Thus, if you wanted to print out each line that contained the word Solaris, you could use the action statement:

```
/Solaris/ { print }
```

The awk program has two special patterns that make report writing easier. These are the BEGIN and END patterns. The action for the BEGIN pattern is automatically run before awk reads any records. The action for END is run after awk processes the last record.

Variables and mathematics

In order for you to do some serious processing, awk allows you to create variables and do mathematics. If you want, you can perform calculations with values found in a specific field. To create a variable, simply assign it a value, like this:

```
var = 5
```

Here, we've assigned the value 5 to the variable *var*. You can treat the fields as numeric values and access them in your calculations.

The awk program provides you with many mathematical operators, such as addition (+), subtraction (-), multiplication (*), and division (/). Putting together what we've learned so far, you could add all the numbers in a column like so:

```
awk 'BEGIN { total=0 } { total += $1 } END {
➥print "Total=",total }' test
```

As you can see in this example, there are three action statements. The first has a pattern of BEGIN, and we use it to set a variable named *total* to 0. The second doesn't have a pattern, so it's executed for every line; this action statement adds the value of the first column to *total*. The third action statement has an END pattern, so awk executes the statement after awk's processed all the records. In this case, the last action statement prints *total*, which at this point contains the total of all values found in the first column in the file *test*.

Decisions and looping

We've already shown how you can execute a pattern based on a decision: Using a pattern like `/Solaris/` tells `awk` to process the action statement only if the word Solaris is found in the record. However, `awk` provides much more sophisticated decision-making capability.

You can use an `if` statement to execute a statement if a certain condition is met. To do so, you use the syntax

```
if (condition)
   stmt1
else
   stmt2
```

Thus, if the specified condition is true, the `if` statement executes the statement labeled *stmt1*. On the other hand, if the statement is false, the `if` statement executes the statement labeled *stmt2*. It will execute one or the other statement, but not both. Please note that the `else` and *stmt2* part of the `if` statement are optional. For example, the statement

```
if ( var == 2 )
   print "Var equals 2"
```

prints "Var equals 2" if, and only if, the value of `var` is 2. Otherwise, it does nothing. Pretty simple, isn't it?

One of the looping constructs is known as a `while` loop. This construction looks like this:

```
while (condition)
   stmt
```

When you get to the `while` statement, `awk` checks the condition. If it's true, then it executes the statement `stmt`. Then `awk` executes the `while` statement again. Thus, as long as the condition is true, the statement will be repeatedly executed. For example, if you execute the statement

```
while ( var > 2 )
   var -= 1
```

when *var* is 10, then `awk` will notice that the condition is true. Then it will subtract one from *var*, leaving it at 9. Then `awk` will execute the `while` statement again leaving *var* set to 8. `awk` will repeatedly execute the `while` statement until finally `awk` subtracts one from *var*, and *var* is less than 2. Then `awk` stops processing the `while` statement and goes on to the next statement.

Compound statements

As you may have noticed, a single statement in a `while` loop isn't terribly useful. There's just enough room for you to change the variable on which your condition is based. You don't really have enough room to do any other work.

As you may expect, there's a way around this: `awk` provides compound statements. In other words, `awk` allows you to treat an arbitrarily large number of statements as a single statement by enclosing them in a pair of curly braces and separating them with semicolons (;). You can put a compound statement anywhere you're allowed to place a statement. As an example, let's take our `while` loop and let it print the value of *var* as it executes. Go ahead and type in the following command:

```
awk 'BEGIN { var=10; while (var>2) { print
➥var; var -= 1 } }'
```

When you press [Enter], you'll see a column of digits from 10 to 3. You won't see a prompt, however. The awk program is waiting for records to process. Since we didn't specify an input file, the program waits for you to enter the records from the keyboard. Simply press [Ctrl]D to tell awk that there aren't any more records.

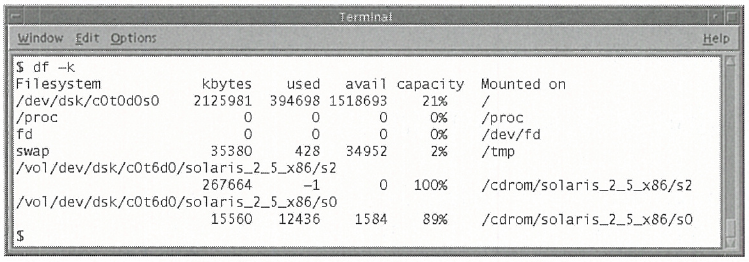
A simple sample

As a real example of the power of `awk`, we'll create a script that will print the amount of disk space on your machine in megabytes and show the total amount of space you've used. To do so, we'll take the output of the `df -k` command, which is close to what we want, convert the values from 1K blocks to megabytes, and format the output into multiple columns with headers.

Let's look at the output we get from the `df -k` command, shown in Figure A on the next page. In manipulating this input with `awk`, we'll look at several useful features of the language. First, the input fields are separated by any amount of white space (spaces or tabs). The output of `df -k` will always have white space separating the fields.

The script we'll use to manipulate and display these values is shown in Figure B. Please

Figure A



```
$ df -k
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t0d0s0    2125981  394698 1518693    21%    /
/proc                      0       0       0     0%    /proc
fd                         0       0       0     0%    /dev/fd
swap                   35380     428   34952     2%    /tmp
/vol/dev/dsk/c0t6d0/solaris_2_5_x86/s2
                      267664      -1       0   100%    /cdrom/solaris_2_5_x86/s2
/vol/dev/dsk/c0t6d0/solaris_2_5_x86/s0
                       15560   12436    1584    89%    /cdrom/solaris_2_5_x86/s0
$
```

This is an example of output from the df -k command. It shows disk usage for mounted file systems in 1K blocks.

Figure B

```
 1) #!/bin/ksh
 2) # Based on df -k, show disk usage in megabytes.
 3) PATH=/bin:/usr/bin
 4) df -k ¦ awk -e '
 5) # Print the column headers
 6) BEGIN {
 7)    printf "%-18s%8s%8s%5s%8s%s\n","File", "Max","Used",
          ➥"Used","Free","Mount"
 8)    printf "%-18s%8s%8s%5s%8s %s\n","System","MByte",
          ➥"MByte","%","MByte","Dir"
 9)    TotalM = TotalUsedM = 0
10)   }
11)
12) # Process each line of df -k output
13) NR >= 2 {
14)    FileSys=$1; MaxK=$2; UsedK=$3; FreeK=$4; MntPt=$6
15)    # Ignore unmounted partitions, i.e., those with 0 byte size.
16)    if ( MaxK > 0 ) {
17)       # If the filename is too long, print it on its own line
18)       if ( length(FileSys) < 19 )
19)          printf "%-18s", FileSys
20)       else
21)          printf "%s\n%18s", FileSys, " "
22)       # Volume is mounted, display stats
23)       MaxM = MaxK/1024; UsedM=UsedK/1024; FreeM=FreeK/1024;
24)       UsedPct = UsedK*100/MaxK;
25)       printf "%8.2f%8.2f%5.1f%8.2f %s\n", MaxM, UsedM, UsedPct,
          ➥FreeM, MntPt
26)       # Accumulate totals
27)       TotalM += MaxM; TotalUsed += UsedM
28)       }
29)    }
30)
31) # Print the ending summary
32) END {
33)    print  "---------------------------------"
34)    printf "Total disk space : %8.2f MBytes\n", TotalM
35)    if ( TotalM > 0 )
36)       printf "  Percent in use : %5.1f%%\n", TotalUsed*100/
          ➥TotalM
37)    }
38) '
```

The dfv script uses awk to calculate and format a table of the megabytes used on all mounted file systems.

note that the line numbers are just for reference, they're not part of the script.

Line 14 of **Figure B** shows how we can assign these field values to variables. Note that assigning them to variables is for clarity and ease-of-use only; we can make calculations and output using the field-number variables directly. Also note that line 16 checks to see if we might divide by 0 (this could occur if a file system listed in */etc/vfstab* isn't mounted). If *MaxK* is zero, we won't process the line any further. Lines 23, 24, 27, and 36 of the *dfv* script show how we can use variables in calculations and how to assign those values to new variables.

The overall structure of this `awk` program consists of three action statements. The first, beginning on line 6, uses the `BEGIN` pattern to specify the actions we want to perform before any input lines are read or processed. In this case, we're printing our header and zeroing out our *TotalM* and *TotalUsed* variables.

The pattern that selects the second and subsequent lines uses the built-in *NR* variable, which contains the count of the current input record or line. Here, if the *NR* variable is 2 or larger, we process the action statement. This has the effect of skipping the first line output by the `df -k` command, which is a column header, as you can see in **Figure A**. The last action statement, starting on line 32, simply prints the total amount of disk space and the percent used.

Formatting with printf

We use the `printf` commands in lines 7 and 8 to format our column headers. In line 25, we use another `printf` statement to display the results of our calculations. In the `printf` commands, a pattern beginning with a percent sign (%) describes the formatting of each field. After defining the output pattern, the `printf` routine substitutes the input values into the pattern in the order they appear.

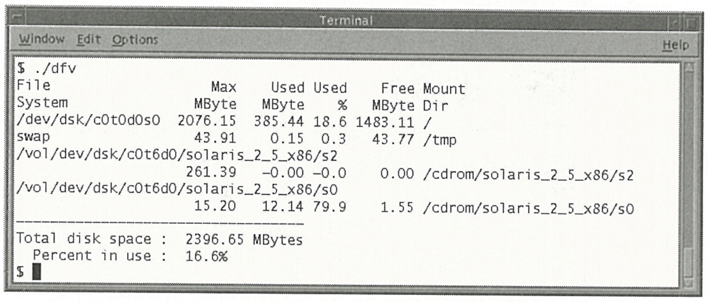
In the header commands, each field is defined as some number of string (`s`) characters. The minus sign (-) preceding the first and last fields indicates that the field will print left-justified instead of the default, right-justified. Therefore, the format string `%8s` specifies a right-justified string that takes eight columns.

In lines 7 and 8, we described all fields using the `s` character, and we gave all input values as character strings. In line 25, only the first and last fields are strings; the rest are either base-10 (decimal) numbers (d) or floating point numbers (f).

As with the string values, the number given is the number of character places in the output format. For floating point fields, the number to the left of the period indicates the total width of the output field, while the number to the right of the period indicates how many of those characters should be reserved for digits to the right of the decimal point. We used the same field widths in our column headers as our numeric output to keep our columns aligned.

Note the `printf` statement in line 36. All the text appearing outside of a field definition is printed literally. In this case, `awk` will print *"Percent in use : "* followed by a number five digits wide (one digit after the decimal point), followed by a % symbol. Note that we had to use two % symbols to get a single one, since the % symbol tells `printf` to start a field definition. Figure C shows what the output of the *dfv* script looks like on one of our machines.

Figure C



```
$ ./dfv
File                   Max    Used Used     Free Mount
System               MByte   MByte    %    MByte Dir
/dev/dsk/c0t0d0s0  2076.15  385.44 18.6  1483.11 /
swap                 43.91    0.15  0.3    43.77 /tmp
/vol/dev/dsk/c0t6d0/solaris_2_5_x86/s2
                    261.39   -0.00 -0.0     0.00 /cdrom/solaris_2_5_x86/s2
/vol/dev/dsk/c0t6d0/solaris_2_5_x86/s0
                     15.20   12.14 79.9     1.55 /cdrom/solaris_2_5_x86/s0
----------------------------------
Total disk space :  2396.65 MBytes
  Percent in use :  16.6%
$ 
```

This is the output of the dfv *script in Figure B when applied to the same file systems as shown in Figure A.*

Conclusion

We didn't show you everything about `awk`. It's too big and complex a language. We do hope that we got your curiosity going, though. The basics are pretty easy to learn, and you can use them to do some fairly complex jobs. As we mentioned, you'll want to read the `man` page to get some more detail on `awk`. If you'd like some more detailed information on the `awk` language, you might want to refer to the following books:

The AWK Programming Language
Alfred V. Aho, Brian W. Kernighan, & Peter J. Weinberger
Addison-Wesley Pub. Co., 1988
ISBN 0-201-07981-X

sed & awk
Dale Dougherty
O'Reilly & Associates, Inc., 1990
ISBN 0-937175-59-5 ❖

PROGRAMMING LANGUAGES

PERL of the Internet

by Brian Schaffner

Are you building a Web server for your office? If so, you'll need to make a lot of decisions about your content on the World Wide Web and how you present it for both external and internal users.

The Web seems to evolve more every day. Interactive Web-page design is one of the hottest areas of evolution on the Internet. There are many tools you can use to manage and manipulate a user's experience at your Web pages. In this article, we'll look at PERL (Practical Extraction and Report Language), a useful tool for CGI (Common Gateway Interface) scripts.

Overview

PERL is an easy-to-use programming language for creating compact programs. Much of the information exchanged on the Internet is in some sort of text format. Even VRML (Virtual Reality Modeling Language) files are sent as text descriptions. PERL is a good tool for Internet programming because it can extract and create reports from text-based data sets (such as VRML files and Web pages).

We'll begin by looking at one of the easiest topics in PERL—its data typing or lack thereof. Next, we'll talk about PERL's extensive set of functions for matching patterns in text strings. Finally, we'll show you how PERL makes it easy to use files for both input and output.

PERL types

PERL, unlike C++ and PASCAL, doesn't require that you strictly typecast your variables. In PERL, a variable can just as easily contain text or integer values. There are essentially four basic data types in PERL: handles, scalar variables, integer-indexed arrays, and string-indexed arrays, as shown in Table A.

A handle data type lets you access files and directories. (We won't go into detail about handles in this article.)

Scalar variables hold numeric or string values. An example of a scalar value assignment would be

```
$myscalar = "Hello!";
```

This assigns the value `Hello!` to the variable `$myscalar`. Notice that the scalar variable begins with a dollar sign. All scalar variables are prefixed with a dollar sign, which tells PERL that the value is a scalar value.

Arrays are sets, or lists, of scalar values. Integer-indexed arrays have a subscript, or index, value that is an integer. An example of an integer-indexed array would be

```
@myarray = ('one', 'two', 'three', 'four');
```

This array assigns the values `one`, `two`, `three`, and four to the array `@myarray`. Notice the `@` prefix for the array. When you refer to an entire list of values, you use the `@` prefix. If you want to use only a single value from an array, you use the `$` prefix to designate the value as a scalar value, such as

```
$mynewscalar = @myarray[1];
```

This statement assigns the value found in `@myarray` at index 1 (the value `two`, because array indexes start at 0, not 1) to the scalar variable `$mynewscalar`.

String-indexed, or associative, arrays are similar to integer-indexed arrays. The difference is that integer-indexed arrays are automatically indexed. With string-indexed arrays, you must explicitly assign index values for each item in the array. An example of a string-indexed array would be

```
%threewishes = ('first', 'new car', 'second',
'new house', 'third', 'three more wishes');
```

This statement assigns three items to the array called `%threewishes`. The first item value is `new car` and its associated index value is `first`. Notice the percent sign in front of the array name. String-indexed arrays carry the percent prefix. As with integer-indexed arrays, you still use a dollar sign when using a single item from the list. However, you use braces (curly brackets) instead of square brackets to enclose index values. For example, the following statement assigns the value of the second item of the `%threewishes` array to a new scalar variable:

```
$newscalar = $threewishes{'second'};
```

Table A: *PERL's four basic data types*

Data type	Data value
Handles	Use to access files and directories
Scalar variables	Values such as strings or integers
Integer-indexed arrays	A list of scalar values with an integer index
String-indexed arrays	A list of scalar values with a string index

A real matchmaker

While PERL data types are easy to use and flexible, the real beauty of PERL lies in its built-in library of pattern-matching functions. The latest release of PERL (version 5.1) includes more than 30 functions for matching patterns in text strings.

The basic PERL matching function provides incredible flexibility. We'll call this basic matching function *m//* because its structure is based on those characters. As an example of this function, let's test whether the string *"Subject: IMPORTANT! This is urgent."* contains the word *IMPORTANT* (in all uppercase letters). To do this, we'll use the code

```
$mystring = "Subject: IMPORTANT! This is
urgent.";
$_ = $mystring;
if (m/(.*)IMPORTANT(.*)/) {
   print "The word IMPORTANT was found.";
}
else {
   print "The word IMPORTANT was not found.";
}
```

The first part of the code simply assigns a value to a scalar variable and then stores the value of the scalar variable to `$_`, which can be thought of as the default variable for many

PERL functions, including text-matching functions. Next, the line

```
if (m/(.*)IMPORTANT(.*)/) {
```

controls the actual matching. The `m/` defines everything between it and the next `/` as the match string. The first part of the match string is a wild card (`.*`). PERL uses the dot as its wild card, which matches any character except a new-line character. The asterisk means that the preceding character can occur zero or more times. Next comes the word we're searching for, and the search string is rounded off with another wild card.

You can use many more parameters for matching patterns in strings. PERL supports parameters for matching digits, new-line characters, tabs, spaces, or any other ASCII character.

The ins and outs of PERL

PERL's tremendous text-matching features are equaled by its easy-to-use stream-based input/output system. PERL, by default, uses three standard streams: one for input, one for output, and one for data. The standard input, STDIN, is the keyboard. The standard output, STDOUT, is the monitor. The standard error, STDERR, is also the monitor.

You can easily redirect the STDIN and STDOUT streams at files from the command line using the < and > operators. The < operator points to a file (or device) that PERL will use as the standard input stream. For instance, suppose you have a PERL program named TEST.PL. You want to run TEST.PL using standard input arguments from a file named *input.dat*. To do this, you'd use the following code:

```
perl test.pl < input.dat
```

If you wanted to direct the output of TEST.PL to your printer (plugged into the LPT1: printer port), you'd use the following code:

```
perl test.pl < input.dat > lpt1:
```

You can just as easily point the output to a filename.

Notes

PERL is an interpreted language, meaning that the PERL program must interpret each line of code before your computer can use it. This is different from compiled programs, which run on a computer without an interpreter. Large programs run faster and more efficiently if they're first compiled. Since PERL programs aren't compiled, it's a good idea to keep them small and manageable so you can run them with a reasonable degree of efficiency.

Conclusion

In this article, we've given you a brief overview of PERL and what it can do. While PERL may at first seem frightening to those not familiar with UNIX programming, it offers tremendous usability once you become accustomed to its syntax. By using PERL, you can easily create programs for text input and output, such as CGI scripts for World Wide Web applications. ❖

BOOK REVIEW

Multithreaded programming in UNIX

by Marco C. Mason

Like many system administrators, you probably have to write small utility programs from time to time. However, most of the programs we write tend to be simple single-threaded programs. That's why we were particularly interested to find the book *Programming with UNIX Threads* at our local bookstore.

If you've ever wanted to write a program that could do multiple things at the same time, or were just curious about how it's done, then you may want to investigate this book. *Programming with UNIX Threads* assumes that you're a proficient C programmer. It doesn't waste any trees giving you an introduction to pointers or hex numbers. Instead, it immediately jumps into the meat of programming with threads. The first two chapters cover the theory behind UNIX threads and how they work. Chapter 3 shows how to convert the theory into practice. After reading this far, we were able to create the program *threads.c*, shown in Figure A.

As you can see, this program doesn't really do much of anything. When you compile and run it, it simply starts a separate thread for each command-line argument. When we create each thread, we give it a random number of iterations to process. These threads simply go to sleep for a random amount of time, decrement their iteration count, and when the count reaches zero, terminate. Figure B shows an example of this program after we executed it with the parameters Apple, Orange, and Banana.

You can start to do some work with multithreaded programs after reading only three chapters. However, if your programs use threads that share some data areas, as we do in our demonstration program, you have to be careful. If you don't find a way to share the data safely, then one thread could corrupt data that's being used by another thread.

Chapters 4 through 10 address synchronization, which allows multiple threads to coordinate their efforts. The author presents several chapters on synchronization for two primary reasons: Each chapter presents a different standard synchronization method, so you may as well get used to these methods. Second, each method has its own strengths and weaknesses. You should explore them all so that you can select the one that best suits the problem.

The book also covers communications between the threads, via explicit signals as well as the synchronization techniques mentioned

Figure A

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

struct tagThread
   {
   pthread_t  pthDscr;  /* Thread descriptor */
   char      *pcName;   /* Thread's name     */
   int        iCount;   /* Iteration count   */
   };
typedef struct tagThread Thread;

/* Sleep for a few seconds, print a message, update the
** loop count, and exit when loop count is 0.
** inp:   pvArg - Pointer to the Thread structure
*/
void *pvThreadProcess( void *pvArg )
   {
   Thread *pThis = (Thread*)pvArg;
   while ( pThis->iCount )
      {
      sleep( rand()/2650 );
      printf("%s: %d passes left\n", pThis->pcName,
             pThis->iCount-- );
      }
   printf("%s: completed\n",pThis->pcName);
   pthread_exit(0);
   }

/* Create a thread for each command-line argument, and
** then wait for them all to terminate.
** inp: argc - # args (# threads to create)
**      argv - arg list (thread names)
*/
int main( int argc, char *argv[] )
   {
   int i, flActThr;

   /* Create an array to hold thread descriptors */
   Thread **ppThreads = (Thread**)
                 calloc(argc, sizeof(Thread*) );
   if ( !ppThreads )
      {
      puts("Can't allocate RAM, sorry!");
      exit(EXIT_FAILURE);
      }

   /* Let's start a thread named for each argument */
   for ( i=1; i<argc; i++ )
      {
      /* Allocate a Thread descriptor */
      Thread *pThread=malloc(sizeof(Thread));
      ppThreads[i] = 0;
      if ( !pThread ) continue;

      /* initialize the thread's data fields */
      pThread->pcName = strdup( argv[i] );
      if ( !pThread->pcName ) continue;
      pThread->iCount = rand() % 5 + 1;

      /* Start the thread */
      if ( !pthread_create( &pThread->pthDscr, NULL,
                pvThreadProcess, (void*)pThread ) )
         ppThreads[i] = pThread;
      }

   /* Check for active threads every 5 seconds */
   do {
      sleep(5);
      for ( i = flActThr = 0; i<argc; i++ )
         if ( ppThreads[i] )
            flActThr |= ppThreads[i]->iCount;
      } while ( flActThr );
   puts("All threads complete.");
```

This program demonstrates just how simple multithreaded programming can be.

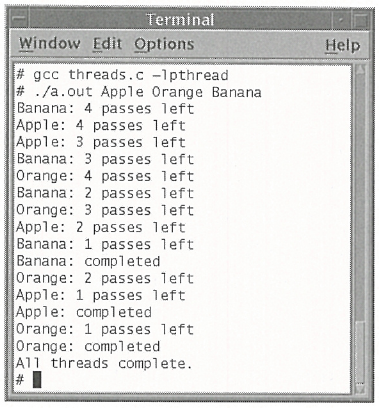
previously. It also discusses scheduling and priorities to help you understand how UNIX decides what thread will execute next.

At the end of the book, the author wraps all these concepts into a demonstration program, ADAM. He intends ADAM to be a framework on which you can build larger multitasking applications. However, ADAM is interesting on its own as an example of just how the concepts of the book fit together.

One aspect of the book that you'll really like is that it's full of examples and notes. Whenever a concept is presented, you can expect to see an example of it in use within a few pages. If you want to learn about multithreaded programming in UNIX, reading *Programming with UNIX Threads* is a must. ❖

Programming with UNIX Threads *by Charles J. Northrup, John Wiley & Sons, Inc., ISBN 0-471-13751-0*

Figure B



```
# gcc threads.c -lpthread
# ./a.out Apple Orange Banana
Banana: 4 passes left
Apple: 4 passes left
Apple: 3 passes left
Banana: 3 passes left
Orange: 4 passes left
Banana: 2 passes left
Orange: 3 passes left
Apple: 2 passes left
Banana: 1 passes left
Banana: completed
Orange: 2 passes left
Apple: 1 passes left
Apple: completed
Orange: 1 passes left
Orange: completed
All threads complete.
#
```

Our demonstration program interleaves execution of three separate threads.

SHELL TOOL TECHNIQUES

Changing the title bar of an xterm window

by Marco C. Mason

If you use OpenWindows, you've probably had lots of Command Tool and Shell Tool windows open at the same time. If you have enough of them on the screen, you can get pretty confused. The more windows you have open, the less of each window you can see before they overlap. Which one are you using for which project? Figure A shows the windows overlapping with only the title bars showing.

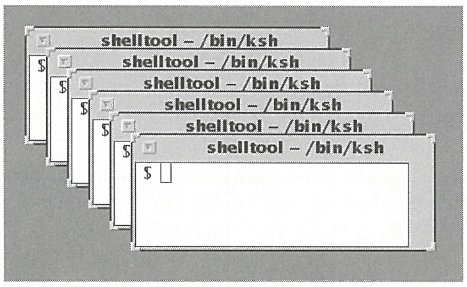
As you can see, all the title bars show the same information: shelltool - /bin/ksh. The title bar is supposed to be informative, but it's telling us only that it's a Shell Tool window running */bin/ksh*. The Command Tool and Shell Tool programs, among others, both use an `xterm` window for I/O. In this article, we'll show you how to change the title bar for an `xterm` window to anything else you want.

Changing the text of an xterm window

It turns out that an `xterm` window treats the title bar and the normal window area as two different windows. The `xterm` provides a special character sequence you can use to put text in the alternate window (i.e., the title bar) and another character sequence to tell the `xterm` when to go back to the original window.

The sequence of characters you use to tell the `xterm` to start putting text in the title bar is [Esc],], then l. You then may send it the text you want on the title bar. When you're finished, send the `xterm` an [Esc] character followed by \.

Figure A



While it's handy to have the ability to use multiple windows to do your job, it can be confusing.

For example, if you use OpenWindows, then you can start a Shell Tool and type the following command:

```
$ echo "^[]lNew Titlebar Text^[\\"
```

When you press the [Esc] key, the `xterm` window displays `^[`. Also, the shell interprets the `\` character as an escape, which tells it to treat the next character as if it had no special meaning. In order to send the `\` character to the `echo` command, you need to use two `\` characters: The first tells it the next character has no special meaning; the second is the character itself.

A quick shell script

Actually remembering and typing the appropriate escape sequences is no great hardship. However, it's simpler to make a shell script do the job for you. The shell script *WinName*, shown in Figure B, name's a window for you.

Keep in mind that `^[` is the [Esc] key, not the keys [Ctrl] and [. If you're using `vi` to enter the shell script, just press [Ctrl]V before pressing the [Esc] key to enter it. One last note: We're comparing the `TERM` environment variable to `sun-cmd` because the `termcaps` entry for an `xterm` is `sun-cmd`.

Figure B

```
if [ $TERM = "sun-cmd" ]; then
  echo "^[]l" $1 "^[\\"
else
  echo "Sorry, you can only use this in a
     ➥sun-cmd window"
fi
```

This short shell script will change the title bar in a sun-cmd (i.e., xterm) window.

If you use this script, you won't have to remember the escape sequences, and you won't have any problems if you mistype something. If you mistype the [Esc]\ part, for example, you'll lose some of your output from successive commands, as the output will go to the title bar. However, the `xterm` doesn't actually change the title bar until it receives the [Esc]\ character sequence, so you won't see a change.

Once you enter the shell script, don't forget to make it executable with the command

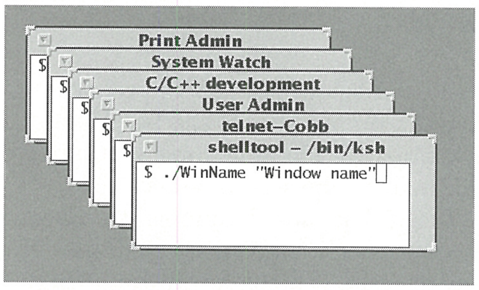
```
$ chmod +x WinName
```

When you use this script, keep this in mind: If you want spaces in your text, enclose the new window title in quotes. Otherwise this script will set the window title to the first word only.

Conclusion

Remember the ugly screen shown in Figure A? Figure C shows the same screen, but with informative title bars. You can use this technique to put any information in the title bar of your `xterm` windows. Once you do so, you'll find it much easier to locate the window you want, since the title bar now describes the window. ❖

Figure C



```
$ ./WinName "Window name"
```

It's much easier to find the window you want when the title bars have useful information in them.

Are you a good tipper?

Do you have any great Solaris tips that you've discovered? If so, send them our way! If we use your tip, it will appear on our weekly online ZDTips service. (Visit www.zdtips.com to check out all our available tip services.) We may also publish it here in *Inside Solaris*. Your byline will appear with the tip, along with your E-mail and/or Web addresses.

Send your tips to inside_solaris@zd.com, fax them to "Solaris tips" at (502) 491-4200, or mail them to

Inside Solaris
The Cobb Group
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220

Turning the File Manager into your command center

by Al Alexander

Many System administrators make a basic mistake about the users of their systems. They believe that most users enjoy working with the obscure intricacies of Solaris as much as they themselves do. It often comes as a shock when they discover that some users are more interested in completing their work than in working on the computer. These users rightly believe that their other work is more important than learning the syntax and command-line options of various Solaris commands.

If you're one of these administrators, you can become an overnight hero when you learn this lesson. Instead of forcing your users to learn the command-line secrets of Solaris, you can find an easier way to bring the power of UNIX to them. One way you can do this is by transforming the seemingly innocent File Manager into a powerful, flexible command center.

Once you make this change, your users will have the ability to run a wide variety of UNIX commands from the File Manager without worrying about any obscure calling conventions. They can, for example, run their CAD and finite-element analysis (FEA) applications, run personal tape backups, manipulate custom batch queues, etc.—all from the confines of the File Manager.

This relatively low-technology solution brings home an important lesson in system administration: No matter how cool and interesting you find the new technologies, the users you support often don't care. It's not vital that you solve every problem with a flashy application. To most users, a simple, reliable way to do their jobs is what's important. In other words, they're looking for a way to make things work better so they'll be more productive.

In this article, we'll show you how you can extend the File Manager by adding new commands, which allows your system's users to do all their work from within the File Manager. Your endeavors may endear you to your users enough so that you receive that coveted "Most Valuable Employee of the Month" award you've been wanting.

Why use the File Manager?

The beauty of the File Manager approach is that for most end users, using a computer system is a file-oriented endeavor. All day long, users open, close, create, copy, delete, modify, and print files (or documents, if you prefer).

If only you could make the File Manager perform other tasks, you'd have the perfect file-oriented tool for your users. Wouldn't it be great if you could simply select a set of files, select Fax, and have the File Manager ask you for the fax number to send the files to?

Obviously, we wouldn't tantalize you without telling you how to accomplish the task, so let's get down to business. First, you need to know that this trick works with the default File Manager used by OpenWindows, but not with the CDE version of File Manager.

Please note that if your users run CDE, then you can customize their environment in other ways to do similar tasks. However, we won't cover those techniques in this article. You can still use our technique, if you want, by running the OpenWindows version of File Manager under CDE. To do so, you should execute the program */usr/openwin/bin/filemgr*.

Locating the custom commands menu

Solaris' File Manager for OpenWindows lets you create your own commands with the support of the Custom Commands option. Once you've defined them, custom commands are available to users through the File menu. You can see this in operation by starting the File Manager, clicking the File menu button, and selecting the Custom Commands option. The resulting list shows you which custom commands you currently have configured for your system. The default application available from the custom commands menu is a UNIX Shell.

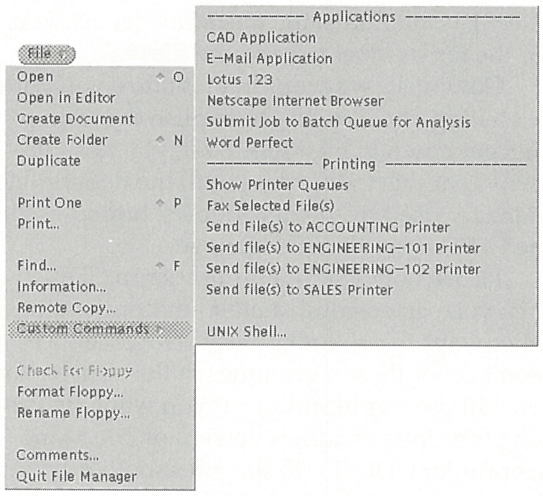
In **Figure A** on the next page, we've created a sample menu to show you some of the things you can do with the Custom Commands option. As you see, you can add a large variety of commands. Please note that the UNIX Shell... entry is always on the Custom Command menu. If you invoke this option, it simply opens a Shell Tool window for you. Now let's see how to create our own custom commands.

Adding a custom command

It's very simple to add custom commands to the File Manager. We'll demonstrate the process by creating a command that will allow users to print one or more files to a printer named Accounting.

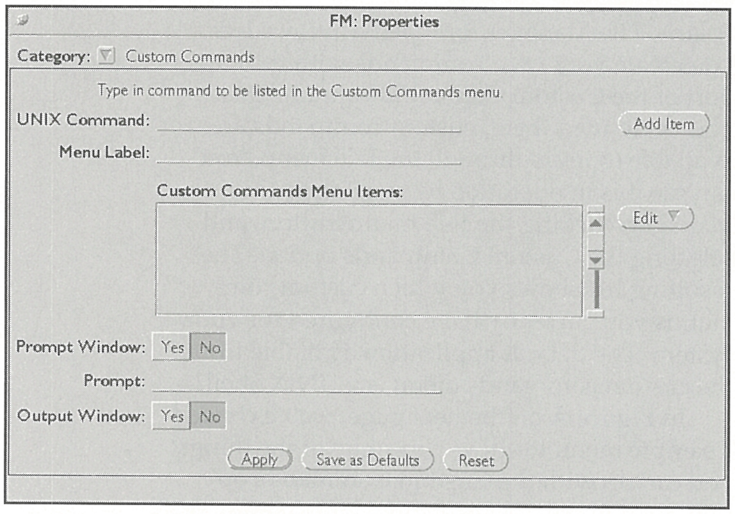
To begin, start the File Manager. Then click the Edit button, and select Properties... from the Edit menu. When you do so, you'll see the FM: Properties dialog box. At the top of the dialog

Figure A



This sample menu shows the flexibility available when you use the Custom Commands option.

Figure B



You can create new commands for File Manager with the FM: Properties dialog box.

box, you'll see a dropdown menu labelled Categories. Click the down arrow and select Custom Commands. When you do so, the FM: Properties dialog box should look like the one shown in Figure B.

You'll use this screen to define your new custom commands. Next, we'll describe all the items in this dialog box and simultaneously build our new custom print command.

Choosing a command

The first field on the Custom Command section of the FM: Properties dialog box is the UNIX Command field. You use this field to enter the command you want the custom command to execute. You may enter any command or script file in this field, along with any parameters you want to use.

Let's begin with our new print command. Normally, when you want to print files to the Accounting printer, you execute the command

```
lp -dAccounting FileList
```

replacing `FileList` with the list of files you want to print. Similarly, if you want to print on a special form, you'd add the form name with the `-f` option. That's simple enough. But where do we get the list of files?

The $FILE token

The File Manager defines a special token, named `$FILE`, that you use to access the list of currently selected files. To do so, you simply add the `$FILE` token in the UNIX Command field in the position where you want the file list to appear. When the File Manager executes your custom command, it will replace the `$FILE` token with the entire list of files you've selected.

The $ARG token

Please note that the File Manager provides another token for your custom commands: the `$ARG` token. The File Manager allows you to prompt the user for special options for your custom command, as we'll discuss in the "Prompting The User" section. When you use the `$ARG` token, the File Manager replaces it with the user's response *before* it executes the custom command.

For our new print command, we'll simply replace `FileList` with `$FILE`. In addition, since the user may want some special print options, we'll add the `$ARG` token to allow this. To get started, click on the UNIX Command field to select it. Now type the command you want to execute. Since we want our custom command to print a file to the Accounting printer, with the possibility of incorporating some options, we'll enter the command

```
lp $ARG -dAccounting $FILE
```

Now suppose the user highlighted two files named *list.c* and *list.h* and then invoked this custom command. Let's also suppose that the File Manager prompted the user for extra options, and the user replied -fLEGAL. In this case, the File Manager would replace the $FILE token with *list.c list.h*, then replace the $ARG token with -fLEGAL, and execute the command as follows:

```
lp -fLEGAL -dAccounting list.c list.h
```

Remember that the $FILE and $ARG tokens are case-sensitive. If you use any lowercase letters, the File Manager won't recognize them. For example, if you use $File instead of $FILE, then the custom command we set up will try to print a file named *$File*, no matter what files you've selected.

Since the File Manager is file-oriented, it expects custom commands to have an embedded $FILE token. If you haven't placed the $FILE token in your custom command, or if you've misspelled it, the File Manager will alert you with the message box shown in Figure C when you click the Add Item button.

If you don't intend for your custom command to use the $FILE token, just press the Apply button. If you want the File Manager to add $FILE to the end of your custom command, press the Add $FILE button. Otherwise, press the Cancel button and fix any problems with your custom command.

Naming your custom command

Now that we've told the File Manager what the custom command should do, we need to give it a label so the user can find it. The File Manager provides the Menu Label field to allow you to give your custom command a name. To do this, click on the Menu Label field and enter the command name; in this case, we'll enter *Print to Accounting*.

Prompting the user

While discussing the $ARG token, we alluded to the fact that you can have the File Manager prompt the user for more information when the user executes the custom command. You can use this feature by setting the Prompt Window field to Yes, then entering the prompt text you want the user to see in the Prompt field.

When the user executes a custom command that has a prompt, the File Manager will create a new window to prompt the user with your custom prompt string. Then the File Manager will replace the $ARG token with the text you enter and the $FILE token (if present) with the list of selected files. Finally, the File Manager will execute the custom command.

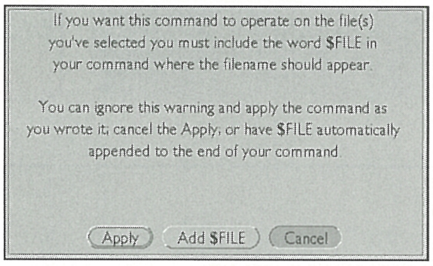
In our example, we used the $ARG token in our custom command, so we should prompt the user for some additional information. To do so, set the Prompt Window field to Yes, then in the Prompt field, enter *Printer Options* to allow the user to specify any additional printer options for the job. When the user executes this custom command, the File Manager will present the Print To Accounting dialog box shown in Figure D.

The output window

For many of your custom commands, you may choose to display the output of the UNIX command. You can do so by setting the Output Window field to Yes in the FM: Properties dialog box. Then, any output from the UNIX command will appear in a separate window that resembles a Shell Tool window. To close the output window, select it (typically by clicking the mouse inside the window) and press the [Return] key.

For our example command, we don't use the output window. So, click the No button

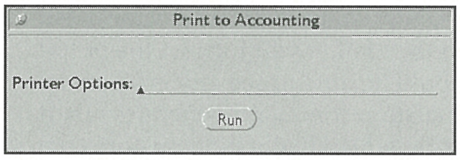
Figure C


If you want this command to operate on the file(s) you've selected you must include the word $FILE in your command where the filename should appear.

You can ignore this warning and apply the command as you wrote it, cancel the Apply, or have $FILE automatically appended to the end of your command.

If you create a custom command without including the $FILE token, the File Manager alerts you with this dialog box.

Figure D



Your custom commands can tell the File Manager to prompt the user for further information on a command.

PERIODICALS MAIL

SunSoft Technical Support

(800) 786-7638

```
*********************3-DIGIT 480
7661905 2096 02/99
```
[redacted]
```
CLINTON TOWNSHIP MI 48035-4218
```

Please include account number from label with any correspondence.

next to the Output Window field to ensure that we don't accidentally open one.

However, when you're creating a custom command and it's giving you trouble, you may want to use the output window as a debugging aid. Many UNIX commands emit error messages when they fail. If you turn on the output window, you'll be able to see any error messages your custom command generates when it tries to run. So remember, if your custom command fails, turn on the output window and examine any output it may provide.

Adding your custom command to the menu

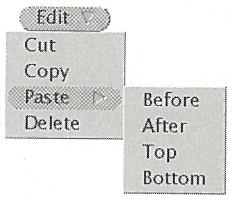
When you create your custom command, you need to add it to the Custom Commands Menu Items box. To do so, click the Add Item button along the right side of the dialog box.

Adding the custom command to the Custom Commands Menu Items box doesn't tell the File Manager to begin using your new custom command. To use these custom commands in the current File Manager session, you must press the Apply button, which tells the File Manager to forget any custom commands it previously stored and load new ones from the Custom Commands Menu Items box.

If you click the Save As Defaults button, you're telling the File Manager to save these custom commands in a special file. From now on, each time you start the File Manager, it will load your custom commands. If you don't click the Save As Defaults button, then the next time you start the File Manager, it will reuse the previous set of custom commands.

Since we want to use our new custom command *immediately*, go ahead and press the Apply button. We also want to save the command for future File Manager sessions, so press the Save As Defaults button, too. If you're the cautious type, you may want to press the Apply button, then test your custom command before actually saving it permanently.

Figure E



You can remove and change the order of the Custom Commands menu with the Edit pulldown menu.

At this point, your new print command should now be available from the Custom Commands submenu on the File menu. To check this, click the File button to pull down the File menu, then select the Custom Commands option. When you do so, you should see the menu label Print to Accounting.

To use your new command, first select a file (or files), then select the Print to Accounting item. When you do so, the File Manager will prompt you with the dialog box shown in Figure D. Once you respond and press the Run button, File Manager will send the selected files with the specified options to the Accounting printer.

Changing your custom command list

Take another look at Figure B. See the Edit button to the right of the Custom Commands Menu Items list box? This pulldown menu allows you to delete and rearrange items in the Custom Commands Menu Items list box. This menu provides the options Cut, Copy, Paste, and Delete. The Paste submenu provides four variations: Before, After, Top, and Bottom, as you can see in Figure E. You can use these options just as you'd imagine. The Paste submenu lets you decide where to paste any custom command.

Please note that you should use these edit options sparingly. It was our experience on all the systems on which we tried these options that the File Manager would easily crash if we overused these options.

Conclusion

Creating custom commands lets you turn your File Manager into a useful command center for your users. Once modified, the File Manager can become the end users' primary interface to the computer system and network, allowing them to launch UNIX commands, custom utilities, and complete applications from one location. ❖

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