Bourne Shell Programming

Andrew Arensburger

March 3, 2000

1 Justification

Why bother to learn shell programming, when you could be out there rollerblading or trying to get a date?

Because, in a word, it's useful.

Many standard utilities (*rdist*, *make*, *cron*, etc.) allow you to specify a command to run at a certain time. Usually, this command is simply passed to the Bourne shell, which means that you can execute whole scripts, should you choose to do so.

Lastly, Unix[1] runs Bourne shell scripts when it boots. If you want to modify the boot-time behavior of a system, you need to learn to write and modify Bourne shell scripts.

2 What's it all about?

First of all, what's a shell? Under Unix, a shell is a command interpreter. That is, it reads commands from the keyboard and executes them.

Furthermore, and this is what this tutorial is all about, you can put commands in a file and execute them all at once. This is known as a script (see Slide 2).

[1] Unix is a registered trademark of The Open Group.

A Simple Script

```
#!/bin/sh
# Rotate procmail log files
cd /homes/arensb/Mail
rm procmail.log.6       # This is redundant
mv procmail.log.5 procmail.log.6
mv procmail.log.4 procmail.log.5
mv procmail.log.3 procmail.log.4
mv procmail.log.2 procmail.log.3
mv procmail.log.1 procmail.log.2
mv procmail.log.0 procmail.log.1
mv procmail.log procmail.log.0
```

There are several things to note here: first of all, comments begin with a hash (`#`) and continue to the end of the line (the first line is special, and we'll cover that in just a moment).

Secondly, the script itself is just a series of commands. I use this script to rotate log files, as it says. I could just as easily have typed these commands in by hand, but I'm lazy, and I don't feel like it. Plus, if I did, I might make a typo at the wrong moment and really make a mess.

3 #!/bin/sh

The first line of any script must begin with `#!`, followed by the name of the interpreter[2]

A script, like any file that can be run as a command, needs to be executable: save this script as *rotatelog* and run

```
chmod +x rotatelog
```

to make it executable. You can now run it by running

```
./rotatelog
```

Unlike some other operating systems, Unix allows any program to be used as a script interpreter. This is why people talk about "a Bourne shell script" or "an awk script." One might even write a *more* script, or an *ls* script (though the latter wouldn't be terribly useful). Hence, it is important to let Unix know which program will be interpreting the script.

When Unix tries to execute the script, it sees the first two characters (`#!`) and knows that it is a script. It then reads the rest of the line to find out which program is to execute the script. For a Bourne shell script, this will be `/bin/sh`. Hence, the first line of our script must be

[2]Some versions of Unix allow whitespace between `#!` and the name of the interpreter. Others don't. Hence, if you want your script to be portable, leave out the blank.

```
#!/bin/sh
```

After the command interpreter, you can have one, and sometimes more, options. Some flavors of Unix only allow one, though, so don't assume that you can have more.

Once Unix has found out which program will be acting as the interpreter for the script, it runs that program, and passes it the name of the script on the command line. Thus, when you run `./rotatelog`, it behaves exactly as if you had run `/bin/sh ./rotatelog`.

4 Variables

`sh` allows you to have variables, just like any programming languages. Variables do not need to be declared. To set a `sh` variable, use

VAR=*value*

and to use the value of the variable later, use

$VAR

or

${VAR}

The latter syntax is useful if the variable name is immediately followed by other text:

$VAR and ${VAR}

```
#!/bin/sh
COLOR=yellow

echo This looks $COLORish
echo This seems ${COLOR}ish
```

```
% ./color
This looks
This seems yellowish
```

There is only one type of variable in `sh`: strings. This is somewhat limited, but is sufficient for most purposes.

4.1 Local vs. environment variables

A `sh` variable can be either a local variable or an environment variable. They both work the same way; the only difference lies in what happens when the script runs another program (which, as we saw earlier, it does all the time).

Environment variables are passed to subprocesses. Local variables are not.

By default, variables are local. To turn a local variable into an environment variable, use

`export` VAR

Variables

```
#!/bin/sh
NETSCAPE_HOME=/usr/imports/libdata

CLASSPATH=$NETSCAPE_HOME/classes
export CLASSPATH

$NETSCAPE_HOME/bin/netscape.bin
```

Slide 4.1 shows a simple wrapper for a program, *netscape.bin*. Here, *NETSCAPE_HOME* is a local variable; *CLASSPATH* is an environment variable. *CLASSPATH* will be passed to *netscape.bin* (*netscape.bin* uses the value of this variable to find Java class files); *NETSCAPE_HOME* is a convenience variable that is only used by the wrapper script; *netscape.bin* doesn't need to know about it, so it is kept local.

The only way to unexport a variable is to `unset` it:

```
unset VAR
```

This removes the variable from the shell's symbol table, effectively making as if it had never existed; as a side effect, the variable is also unexported[3].

Since you may want to use this variable later, it is better not to define it in the first place.

Also, note that if a variable was passed in as part of the environment, it is already an environment variable when your script starts running. If there is a variable that you really don't want to pass to any subprocesses, you should `unset` it near the top of your script. This is rare, but it might conceivably happen.

If you refer to a variable that hasn't been defined, `sh` substitutes the empty string.

[3] Also, if you have a function by the same name as the variable, `unset` will also delete that function

Unset variables

```
#!/bin/sh
echo aaa $FOO bbb
echo xxx${FOO}yyy
```

```
% ./none
aaa bbb
xxxyyy
```

4.2 Special variables

`sh` treats certain variables specially: some are set for you when your script runs, and some affect the way commands are interpreted.

Special Variables

`$1`, `$2`...`$9`	Command-line arguments.
`$0`	Script name.
`$*`, `$@`	All command-line arguments.
`$#`	Number of command-line arguments.
`$?`	Exit status of last command.
`$-`	`sh` options.
`$$`	PID of the current process.
`$|`	PID of the last background command.
`$IFS`	Input Field Separator.

4.2.1 Command-line arguments

The most useful of these variables are the ones referring to the command-line arguments. $1 refers to the first command-line argument (after the name of the script), $2 refers to the second one, and so forth, up to $9.

If you have more than nine command-line arguments, you can use the `shift` command: this discards the first command-line argument, and bumps the remaining ones up by one position.

The variable \$0 (zero) contains the name of the script (`argv[0]` in C programs).

Often, it is useful to just list all of the command-line arguments. For this, `sh` provides the variables \$* (star) and \$@ (at). Each of these expands to a string containing all of the command-line arguments, as if you had used `$1 $2 $3`...

The difference between \$* and \$@ lies in the way they behave when they occur inside double quotes (see Section 6.3): \$* behaves in the normal way, whereas \$@ creates a separate double-quoted string for each command-line argument. That is, `"$*"` behaves as if you had written `"$1 $2 $3"`, whereas `"$@"` behaves as if you had written `"$1" "$2" "$3"`.

Finally, \$# contains the number of command-line arguments that were given.

4.2.2 Other special variables

\$? gives the exit status of the last command that was executed. This should be zero if the command exited normally.

\$- lists all of the options with which `sh` was invoked. See **sh**(1) for details.

\$\$ holds the PID of the current process.

\$! holds the PID of the last command that was executed in the background.

\$IFS (Input Field Separator) determines how `sh` splits strings into words.

4.3 Quasi-variable constructs

The \${VAR} construct is actually a special case of a more general class of constructs:

\${VAR:-*expression*} Use default value: if VAR is set and non-null, expands to \$VAR. Otherwise, expands to *expression*.

\${VAR:=*expression*} Set default value: if VAR is set and non-null, expands to \$VAR. Otherwise, sets VAR to *expression* and expands to *expression*.

\${VAR:?[*expression* }] If VAR is set and non-null, expands to \$VAR. Otherwise, prints *expression* to standard error and exits with a non-zero exit status.

\${VAR:+*expression*} If VAR is set and non-null, expands to the empty string. Otherwise, expands to *expression*.

\${#VAR} Expands to the length of \$VAR.

The above patterns test whether VAR is set and non-null. Without the colon, they only test whether VAR is set.

5 Patterns and Globbing

`sh` supports a limited form of pattern-matching. The operators are

`*` Matches zero or more characters.

`?` Matches exactly one character.

`[`*range*`]` Matches any character in *range*.

When an expression containing these characters occurs in the middle of a command, `sh` substitutes the list of all files whose name matches the pattern. This is known as "globbing." Otherwise, these are used mainly in the `case` construct (see Section 8.4).

As a special case, when a glob begins with `*` or `?`, it does not match files that begin with a dot. To match these, you need to specify the dot explicitly (*e.g.*, `.*`, `/tmp/.*`).

Note to MS-DOS users: under MS-DOS, the pattern `*.*` matches every file. In `sh`, it matches every file that contains a dot.

6 Quoting

If you say something like

```
echo * MAKE   $$$   FAST *
```

it won't do what you want: first of all, `sh` will expand the `*`s and replace them with a list of all the files in the current directory. Then, since any number of tabs or blanks can separate words, it will compress the three spaces into one. Finally, it will replace the first instance of `$$` with the PID of the shell. This is where quoting comes in.

`sh` supports several types of quotes. Which one you use depends on what you want to do.

6.1 Backslash

Just as in C strings, a backslash ("\") removes any special meaning from the character that follows. If the character after the backslash isn't special to begin with, the backslash has no effect.

The backslash is itself special, so to escape it, just double it: `\\`.

6.2 Single quotes

Single quotes, such as

```
'foo'
```

work pretty much the way you'd expect: anything inside them (except a single quote) is quoted. You can say

```
echo '* MAKE   $$$   FAST *'
```

and it'll come out the way you want it to.

Note that a backslash inside single quotes also loses its special meaning, so you don't need to double it. There is no way to have a single quote inside single quotes.

6.3 Double quotes

Double quotes, such as

```
"foo"
```

preserve spaces and most special characters. However, variables and backquoted expressions (see Section 6.4) are expanded and replaced with their value.

6.4 Backquotes

If you have an expression within backquotes (also known as backticks), *e.g.*,

```
`cmd`
```

the expression is evaluated as a command, and replaced with whatever the expression prints to its standard output. Thus,

```
echo You are `whoami`
```

prints

```
You are arensb
```

(if you happen to be me, which I do).

7 Built-in commands

`sh` understands several built-in commands, *i.e.*, commands that do not correspond to any program. These commands include:

{ *commands* ; },(*commands*) Execute *commands* in a subshell. That is, run them as if they were a single command. This is useful when I/O redirection (see Section 9) is involved, since you can pipe data to or from a mini-script inside a pipeline.

The { *commands*; } variant is somewhat more efficient, since it doesn't spawn a true subshell. This also means that if you set variables inside of it, the changes will be visible in the rest of the script.

: (colon) Does nothing. This is generally seen as

: ${VAR:=*default*}

. ***filename*** The dot command reads in the specified filename, as if it had occurred at that place in the script.

`bg` [***job***], `fg` [***job***] `bg` runs the specified job (or the current job, if none is specified) in the background. `fg` resumes the specified job (or the current job, if none is specified) in the foreground. Jobs are specified as %*number*. The `jobs` command lists jobs.

`cd` [***dir***] Sets the current directory to *dir*. If *dir* is not specified, sets the current directory to the home directory.

`pwd` Prints the current directory.

`echo` [***args***] Prints *args* to standard output.

`eval` ***args*** Evaluates *args* as a `sh` expression. This allows you to construct a string on the fly (*e.g.*, using a variable that holds the name of a variable that you want to set) and execute it.

`exec` ***command*** Runs the specified command, and replaces the current shell with it. That is, nothing after the `exec` statement will be executed, unless the `exec` itself fails.

`exit` [***n***] Exit the current shell with exit code *n*. This defaults to zero.

`kill` [-***sig***] %***job*** Send signal *sig* to the specified job. *sig* can be either numeric or symbolic. `kill -l` lists all available signals. By default, *sig* is SIGTERM (15).

`read` ***name...*** Reads one line from standard input and assigns it to the variable *name*. If several variables *name1*, *name2*, *name3* etc. are specified, then the first word of the line read is assigned to *name1*, the second to *name2*, and so forth. Any remaining words are assigned to the last variable.

`set` [+/-***flag***] [***arg***] With no arguments, prints the values of all variables.

`set -`*x* turns on the *x* option to `sh`; `set +`*x* turns it off.

`set` *args...* sets the command-line arguments to *args*.

`test` ***expression*** Evaluates a boolean expression and exits with an exit code of zero if it is true, or non-zero if it is false. See Section 11.3 for more details.

`trap` [***command sig***]... If signal *sig* is sent to the shell, execute *command*. This is useful for exiting cleanly (*e.g.*, removing temporary files etc.) when the script is interrupted.

`ulimit` Print or set system limits on resource usage.

`umask` [***nnn***] Sets the umask to *nnn* (an octal number). With no argument, prints the current umask. This is most useful when you want to create files, but want to restrict who can read or write them.

`wait` [***n***] Wait for the background process whose PID is *n* to terminate. With no arguments, waits for all of the background processes to terminate.

Bear in mind that the list of builtins varies from one implementation to another, so don't take this list as authoritative.

8 Flow control

`sh` supports several flow-control constructs, which add power and flexibility to your scripts.

8.1 `if`

The `if` statement is a simple conditional. You've seen it in every programming language. Its syntax is

```
if condition ; then
   commands
[elif condition ; then
   commands]...
[else
   commands]
fi
```

That is, an if-block, optionally followed by one or more elif-blocks (`elif` is short for "else if"), optionally followed by an else-block, and terminated by `fi`.

The `if` statement pretty much does what you'd expect: if *condition* is true, it executes the if-block. Otherwise, it executes the else-block, if there is one. The `elif` construct is just syntactic sugar, to let you avoid nesting multiple if statements.

<u>`if`</u>

```
#!/bin/sh
myname=`whoami`

if [ $myname = root ]; then
        echo "Welcome to FooSoft 3.0"
else
        echo "You must be root to run this script"
        exit 1
fi
```

The more observant among you (or those who are math majors) are thinking, "Hey! You forgot to include the square brackets in the syntax definition!"

Actually, I didn't: [is actually a command, */bin/[*, and is another name for the *test* command[4]. See Section 11.3 for the details on the *test* command.

The *condition* can actually be any command. If it returns a zero exit status, the condition is true; otherwise, it is false. Thus, you can write things like

```
#!/bin/sh
user=arnie
if grep $user /etc/passwd; then
        echo "$user has an account"
else
        echo "$user doesn't have an account"
fi
```

8.2 while

The `while` statement should also be familiar to you from any number of other programming languages. Its syntax in `sh` is

```
while condition; do
      commands
done
```

As you might expect, the `while` loop executes commands as long as *condition* is true. Again, *condition* can be any command, and is true if the command exits with a zero exit status.

A `while` loop may contain two special commands: `break` and `continue`.

`break` exits the `while` loop immediately, jumping to the next statement after the `done`.

`continue` skips the rest of the body of the loop, and jumps back to the top, to where *condition* is evaluated.

8.3 for

The `for` loop iterates over all of the elements in a list. Its syntax is

```
for var in list; do
    commands
done
```

list is zero or more words. The `for` construct will assign the variable *var* to each word in turn, then execute *commands*. For example:

```
#!/bin/sh
for i in foo bar baz "do be do"; do
```

[4]This is why you shouldn't call a test program *test*: if you have "." at the end of your path, as you should, executing `test` will run */bin/test*.

```
        echo "\$i: $i"
done
```

will print

```
$i: foo
$i: bar
$i: baz
$i: do be do
```

A `for` loop may also contain `break` and `continue` statements. They work the same way as in the `while` loop.

8.4 `case`

The `case` construct works like C's `switch` statement, except that it matches patterns instead of numerical values. Its syntax is

`case` *expression* `in`
  pattern`)`
    commands
    `;;`
  ...
`esac`

expression is a string; this is generally either a variable or a backquoted command.

pattern is a glob pattern (see Section 5).

The patterns are evaluated in the order in which they are seen, and only the first pattern that matches will be executed. Often, you'll want to include a "none of the above" clause; to do this, use `*` as your last pattern.

9 I/O redirection

A command's input and/or output may be redirected to another command or to a file. By default, every process has three file descriptors: standard input (0), standard output (1) and standard error (2). By default, each of these is connected to the user's terminal.

However, one can do many interesting things by redirecting one or more file descriptor:

< *filename* Connect standard input to the file *filename*. This allows you to have a command read from the file, rather than having to type its input in by hand.

> *filename* Connect standard output to the file *filename*. This allows you to save the output of a command to a file. If the file does not exist, it is created. If it does exist, it is emptied before anything happens.

(Exercise: why doesn't `cat * > zzzzzzz` work the way you'd expect?)

`>>` ***filename*** Connects standard output to the file *filename*. Unlike `>`, however, the output of the command is appended to *filename*.

`<<`***word*** This construct isn't used nearly as often as it could be. It causes the command's standard input to come from... standard input, but only until *word* appears on a line by itself. Note that there is no space between `<<` and *word*.

This can be used as a mini-file within a script, *e.g.*,

```
cat > foo.c <<EOT
#include <stdio.h>

main()
{
printf("Hello, world!\n");
}
EOT
```

It is also useful for printing multiline messages, *e.g.*:

```
line=13
cat <<EOT
An error occurred on line $line.
See page 98 of the manual for details.
EOT
```

As this example shows, by default, `<<` acts like double quotes (*i.e.*, variables are expanded). If, however, *word* is quoted, then `<<` acts like single quotes.

`<&`***digit*** Use file descriptor *digit* as standard input.

`>&`***digit*** Use file descriptor *digit* as standard output.

`<&-` Close standard input.

`>&-` Close standard output.

command1 `|` ***command2*** Creates a pipeline: the standard output of *command1* is connected to the standard input of *command2*. This is functionally identical to

command1 `> /tmp/foo`
command2 `< /tmp/foo`

except that no temporary file is created, and both commands can run at the same time[5].

Any number of commands can be pipelined together.

[5] There is a proverb that says, "A temporary file is just a pipe with an attitude and a will to live."

command1* `&&` *command2 Execute *command1*. Then, if it exited with a zero (true) exit status, execute *command2*.

command1* `||` *command2 Execute *command1*. Then, if it exited with a non-zero (false) exit status, execute *command2*.

If any of the redirection constructs is preceded by a digit, then it applies to the file descriptor with that number, rather than the default (0 or 1, as the case may be). For instance,

command `2>&1 >` *filename*

associates file descriptor 2 (standard error) with the same file as file descriptor 1 (standard output), then redirects both of them to *filename*.

This is also useful for printing error messages:

```
echo "Danger!  Danger Will Robinson!" 1>&2
```

Note that I/O redirections are parsed in the order they are encountered, from left to right. This allows you to do fairly tricky things, including throwing out standard output, and piping standard output to a command.

10 Functions

When a group of commands occurs several times in a script, it is useful to define a function. Defining a function is a lot like creating a mini-script within a script.

A function is defined using

```
name () {
        commands
}
```

and is invoked like any other command:

name args...

You can redirect a function's I/O, embed it in backquotes, etc., just like any other command.

One way in which functions differ from external scripts is that the shell does not spawn a subshell to execute them. This means that if you set a variable inside a function, the new value will be visible outside of the function.

A function can use `return` *n* to terminate with an exit status of *n*. Obviously, it can also `exit` *n*, but that would terminate the entire script.

10.1 Function arguments

A function can take command-line arguments, just like any script. Intuitively enough, these are available through \$1, \$2... \$9 just like the main script.

11 Useful utilities

There are a number of commands that aren't part of `sh`, but are often used inside `sh` scripts. These include:

11.1 basename

`basename` *pathname* prints the last component of *pathname*:

```
basename /foo/bar/baz
```

prints

```
baz
```

11.2 dirname

The complement of `basename`: `dirname` *pathname* prints all but the last component of *pathname*, that is the directory part: *pathname*:

```
dirname /foo/bar/baz
```

prints

```
/foo/bar
```

11.3 [

`/bin/[` is another name for `/bin/test`. It evaluates its arguments as a boolean expression, and exits with an exit code of 0 if it is true, or 1 if it is false.

If `test` is invoked as `[`, then it requires a closing bracket `]` as its last argument. Otherwise, there must be no closing bracket.

`test` understands the following expressions, among others:

`-e` ***filename*** True if *filename* exists.

`-d` ***filename*** True if *filename* exists and is a directory.

`-f` ***filename*** True if *filename* exists and is a plain file.

`-h` ***filename*** True if *filename* exists and is a symbolic link.

`-r` ***filename*** True if *filename* exists and is readable.

`-w` ***filename*** True if *filename* exists and is writable.

`-n` ***string*** True if the length of *string* is non-zero.

`-z` ***string*** True if the length of *string* is zero.

string True if *string* is not the empty string.

s1* = *s2 True if the strings *s1* and *s2* are identical.

s1* != *s2 True if the strings *s1* and *s2* are not identical.

n1* -eq *n2 True if the numbers *n1* and *n2* are equal.

n1* -ne *n2 True if the numbers *n1* and *n2* are not equal.

n1* -gt *n2 True if the number *n1* is greater than *n2*.

n1* -ge *n2 True if the number *n1* is greater than or equal to *n2*.

n1* -lt *n2 True if the number *n1* is less than *n2*.

n1* -le *n2 True if the number *n1* is less than or equal to *n2*.

! *expression* Negates *expression*, that is, returns true iff *expression* is false.

expr1* -a *expr2 True if both expressions, *expr1* and *expr2* are true.

expr1* -o *expr2 True if either expression, *expr1* or *expr2* is true.

(*expression*) True if *expression* is true. This allows one to nest expressions.

Note that lazy evaluation does not apply, since all of the arguments to `test` are evaluated by `sh` before being passed to `test`. If you stand to benefit from lazy evaluation, use nested `if`s.

11.4 echo

`echo` is a built-in in most implementations of `sh`, but it also exists as a standalone command.

`echo` simply prints its arguments to standard output. It can also be told not to append a newline at the end: under BSD-like flavors of Unix, use

```
echo -n "string"
```

Under SystemV-ish flavors of Unix, use

```
echo "string\c"
```

11.5 awk

Awk (and its derivatives, *nawk* and *gawk*) is a full-fledged scripting language. Inside `sh` scripts, it is generally used for its ability to split input lines into fields and print one or more fields. For instance, the following reads */etc/passwd* and prints out the name and uid of each user:

```
awk -F : '{print $1, $3 }' /etc/passwd
```

The `-F :` option says that the input records are separated by colons. By default, *awk* uses whitespace as the field separator.

11.6 sed

Sed (stream editor) is also a full-fledged scripting language, albeit a less powerful and more convoluted one than *awk*. In `sh` scripts, *sed* is mainly used to do string substitution: the following script reads standard input, replaces all instances of "foo" with "bar", and writes the result to standard output:

```
sed -e 's/foo/bar/g'
```

The trailing `g` says to replace all instances of "foo" with "bar" on a line. Without it, only the first instance would be replaced.

11.7 tee

`tee` [`-a`] *filename* reads standard input, copies it to standard output, and saves a copy in the file *filename*.

By default, `tee` empties *filename* before it begins. With the `-a` option, it appends to *filename*.

12 Debugging

Unfortunately, there are no symbolic debuggers such as *gdb* for `sh` scripts. When you're debugging a script, you'll have to rely the tried and true method of inserting trace statements, and using some useful options to `sh`:

The `-n` option causes `sh` to read the script but not execute any commands. This is useful for checking syntax.

The `-x` option causes `sh` to print each command to standard error before executing it. Since this can generate a lot of output, you may want to turn tracing on just before the section that you want to trace, and turn it off immediately afterward:

```
set -x
# XXX - What's wrong with this code?
grep $user /etc/passwd 1>&2 > /dev/null
set +x
```

13 Style

Here follow a few tips on style, as well as one or two tricks that you may find useful.

13.1 Prefer simple, linear application

The advantages of `sh` are that it is portable (it is found on every flavor of Unix, and is reasonably standard from one implementation to the next), and can do most things that you may want to do with it. However, it is not particularly fast, and there are no good debugging tools for `sh` scripts.

Therefore, it is best to keep things simple and linear: do A, then do B, then do C, and exit. If you find yourself writing many nested loops, or building *awk* scripts on the fly, you're probably better off rewriting it in *Perl* or C.

13.2 Put customization variables at the top

If there's any chance that your script will need to be modified in a predictable way, then put a customization variable near the top of the script. For instance, if you need to run *gmake*, you might be tempted to write

```
#!/bin/sh
...300 lines further down...
/usr/local/bin/gmake foo
```

However, someone else might have *gmake* installed somewhere else, so it is better to write

```
#!/bin/sh
GMAKE=/usr/local/bin/gmake
...300 lines further down...
$GMAKE foo
```

13.3 Don't go overboard with functions

Functions are neat, but `sh` is not Pascal or C. In particular, don't try to encapsulate everything inside a function, and avoid having functions call each other. I once had to debug a script where the function calls were six deep at times. It wasn't pretty.

13.4 Multi-line strings

Remember that you can put newlines in single- or double-quoted strings. Feel free to use this fact if you need to print out a multi-line error message.

13.5 Use : ${VAR:=value} to set defaults

Let's say that your script allows the user to edit a file. It might be tempting to include the line

```
vi $filename
```

in your script. But let's say that the user prefers to use *Emacs* as his editor. In this case, he can set `$VISUAL` to indicate his preference.

However,

```
$VISUAL $filename
```

is no good either, because `$VISUAL` might not be set.

So use

```
: ${VISUAL:=vi}
$VISUAL $filename
```

to set `$VISUAL` to a reasonable default, if the user hasn't set it.

14 Paranoia

As with any programming language, it is very easy to write `sh`scripts that don't do what you want, so a healthy dose of paranoia is a good thing. In particular, scripts that take input from the user must be able to handle any kind of input. CGI-bin scripts will almost certainly be given not only incorrect, but malicious input. Errors in scripts that run as root or bin can cause untold damage as well.

14.1 Setuid scripts

DON'T

As we saw in section 3, the way scripts work, Unix opens the file to find out which program will be the file's interpreter. It then invokes the interpeter, and passes it the script's pathname as a command line argument. The interpreter then opens the file, reads it, and executes it.

From the above, you can see that there is a delay between when the OS opens the script, and when the interpreter opens it. This means that there is a race condition that an attacker can exploit: create a symlink that points to the setuid script; then, after the OS has determined the interpeter, but before the interpreter opens the file, replace that symlink with some other script of your choice. Presto! Instant root shell!

This problem is inherent to the way scripts are processed, and therefore cannot easily be fixed.

Compiled programs do not suffer from this problem, since *a.out* (compiled executable) files are not closed then reopened, but directly loaded into memory. Hence, if you have an application that needs to be setuid, but is most easily written as a script, you can write a wrapper in C that simply *exec*s the script. You still need to watch out for the usual problems that involve writing setuid programs, and you have to be paranoid when writing your script, but all of these problems are surmountable. The double-open problem is not.

14.2 `$IFS`

The very first statement in your script should be

```
IFS=
```

which resets the input field separator to its default value. Otherwise, you inherit `$IFS` from the user, who may have set it to some bizarre value in order to make `sh` parse strings differently from the way you expect, and induce weird behavior.

14.3 $PATH

Right after you set `$IFS`, make sure you set the execution path. Otherwise, you inherit it from the user, who may not have it set to the same value as you do.

In particular, the user might have "." (dot) as the first element of his path, and put a program called *ls* or *grep* in the current directory, with disastrous results.

In general, never put "." or any other relative directory on your path.

I like to begin by putting the line

```
PATH=
```

at the top of a new script, then add directories to it as necessary (and only add those directories that *are* necessary).

14.4 Quoted variables

Remember that the expansion of a variable might include whitespace or other special characters, whether accidentally or on purpose. To guard against this, make sure you double-quote any variable that should be interpreted as a single word, or which might contain unusual characters (*i.e.*, any user input, and anything derived from that).

I once had a script fail because a user had put a square bracket in his GCOS field in */etc/passwd*. You're best off just quoting everything, unless you know for sure that you shouldn't.

14.5 Potentially unset variables

Remember that variables may not be set, or may be set to the null string. For instance, you may be tempted to write

```
if [ $answer = yes ]; then
```

However, `$answer` might be set to the empty string, so `sh` would see `if [ = yes ]; then`, which would cause an error. Better to write

```
if [ "$answer" = yes ]; then
```

The danger here is that `$answer` might be set to `-f`, so `sh` would see `if [ -f = yes ]; then`, which would also cause an error.

Therefore, write

```
if [ x"$answer" = xyes ]; then
```

which avoids both of these problems.

15 What about the C shell?

The C shell, *csh*, and its variant *tcsh* is a fine interactive shell (I use *tcsh*), but is a lousy shell for writing scripts. See Tom Christiansen's ariticle, "Csh Programming Considered Harmful" for the gory details.