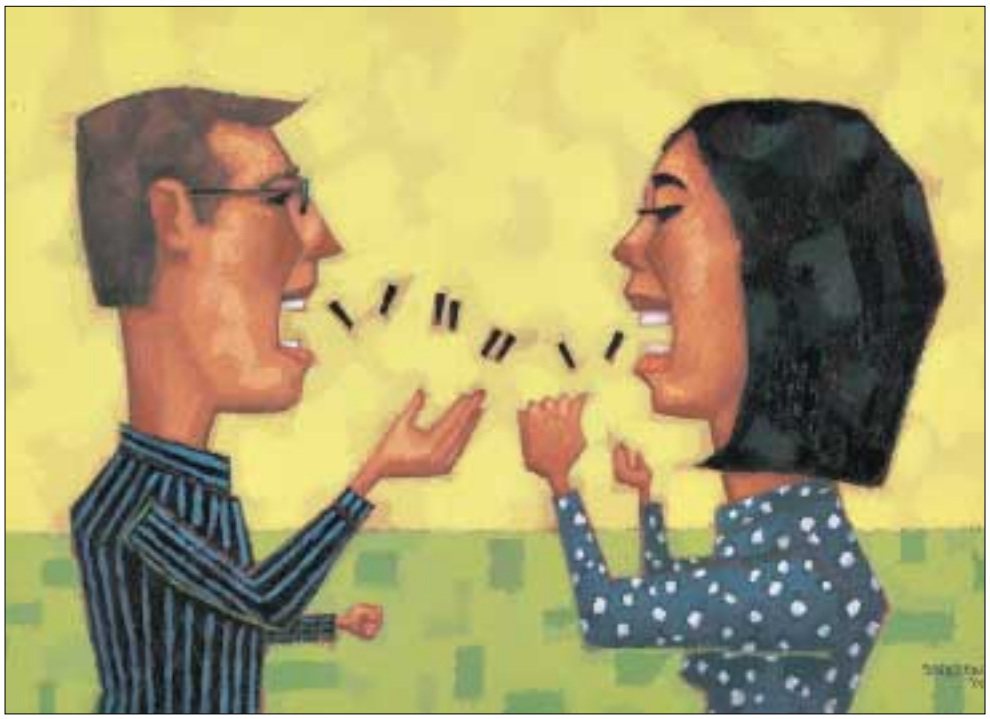
by Peter Collinson, **Hillside Systems**



STEPHEN SCHILDBACH

To Quote or not to Quote

I was engaged in the difficult exercise of deciding what to write this month, when I was saved by Sivaram Neelakantan, who sent me email asking a question about some examples in Kernighan and Pike's excellent book, *The UNIX Programming Environment* (see "Further Reading," Page 27). Sivaram was working through the book and needed a little explanation. The email gave me the chance to pull the book from my shelf and look at it again. For a book written in 1984, it's still completely relevant. If you don't have a copy, then put it on your present list for your next available festival of receiving goodies.

Sivaram's question concerns quoting characters in the shell, making sure the commands you run are supplied with the correct arguments. The topic touches on the way that UNIX systems work, and I think every UNIX user needs some understanding of the mechanics to help clarify why things are the way they are.

UNIX was designed to allow users to invoke commands by typing characters into a program whose job is to launch commands. The program is called the user's "shell" because it's the outer casing of the operating system, providing an interface to allow the user to get their job done. At the time UNIX was designed, it was a curiosity that the user's shell was a normal program with no special privilege. Up to that point in most operating systems, the functions of the shell were provided as part of the operating system, as a top layer of the onion skin, as it were.

UNIX threw away the notion of onion skins and the system is essentially two levels: the kernel, which is resident all the time in the memory of the machine, and above that we have many user processes. The kernel is responsible for controlling the hardware and providing a standard set of interfaces as "system calls" to the user processes. Essentially, the kernel provides support for the way each user process communicates with the outside world. For example, the kernel maintains file system structure. Processes use system calls to access files and are able to deal with them as linear sequences of bytes without worrying about how files are implemented, or where the blocks of each file are actually placed on the disk.

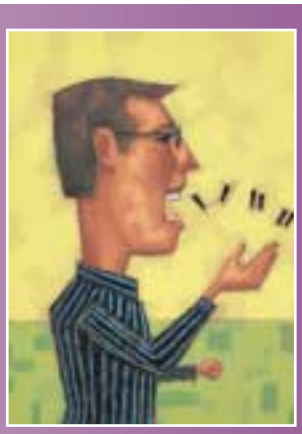
The kernel also provides support for processes, generating an illusion of how the world works that's often called the "process model." A keystone of the model is that processes think they are a single program running in an empty machine. This virtual machine has memory for storing the binary code of the program that makes the process work, and also for storing any data the process may use. Since the beginning, UNIX has been able to share the code sections of processes, so if there are five shells running, there will be only one live copy of their code resident in the machine.

User processes are all "equal," that

is they are all programmed to the same model and compete for system resources equally. As George Orwell points out in *Animal Farm*, it's a feature of human existence that "some are more equal than others." This is true of processes run by the root user, we give such processes the ability to circumvent system security that is provided by the file system.

Creating New Processes

We know that a live UNIX system is a resident kernel and a bunch of running user processes. Any process can use the appropriate system calls to create a new process and cause it to run a new program. The first step is for a process to use the `fork()` system call. It creates two identical processes, both running in parallel in the machine. There's a slight difference between the two processes: the process that invoked the `fork()` call, the *parent*, is told the process ID of the new process when the `fork()` call returns, while the new process, the *child*, is given a zero return value from `fork`. Because both the parent and the child are running the same code, the difference in the returned value allows the programmer to create code that will be executed in only one of the two processes.



Like many other programs, UNIX shells read a line of text typed in by the user, do something with the text and then loop waiting for another line.

Actually, on some occasions, the `fork()` call is the end of the story. There are several applications where a program wants to replicate itself. For example, Web servers often run several instances of themselves to ensure that there is always a process listening for a new request for data.

If the process wants to start a completely different program running, then the child portion of its code will use the `exec()` system call to inform the kernel of this desire. The system libraries provide several flavors of interface for the `exec()` call, if you are interested, then

```
$ man -s 2 exec
```

will supply the complete story. I'm avoiding the full gritty details here for simplicity. The basic `exec` system call has several arguments. The first argument is the pathname of the file that contains the binary of program that is to be loaded. The remaining arguments are text strings that are passed into the running program when it is started by the kernel.

The file that contains the binary that is to be loaded must have its permissions set up to allow execute access for the user that owns the process doing the `exec()` call. If the permissions are OK, then the kernel will dismantle the process image that's using the `exec()` call and will load the new file into memory. The new process will have the same process ID as before, and will inherit several aspects of the process model, such as extant open files for its standard input, output and error channels.

Immediately before the new process begins running, the program arguments from the `exec` call are placed into the new process in a known place so that it may access them if it wishes. The process sees a count of the total number of arguments that have been passed, named in most literature as `argc`, and an array of strings, named `argv`. This allows us to pass text strings from parent to child, and for the child to interpret them in any way that makes sense to the programmer. Incidentally, the first string is conventionally the name of the command being invoked. Because the argument strings are accessed by an array, and arrays in the C language are numbered from zero, programmers think of this as the "zeroth" argument.

Shells

OK. We have a mechanism to create new processes and the ability to pass argument strings into the new running program. How does a shell use this? As I said, a shell is not a special program. Like many other programs, UNIX shells read a line of text typed in by the user, do something with the text and then loop waiting for another line. In the simplest case, shells treat the text as a command name and a set of arguments to that command. Interpretation of the text is done using a set of inbuilt rules. Once the command has been determined, the shell will `fork` to generate a new copy of itself, create all the arguments for the `exec` system call and start the process the user requested.

Generally, the parent shell will wait patiently for the new child process to finish before attempting to read another line from the terminal. I say "generally" because if we end the input line with an ampersand, then the parent doesn't wait. Once the `fork()` has been called, and the child is busily setting itself up, the parent will read another line from the terminal.

I said that when you type the line of input, the shell interprets it according to its own set of inbuilt rules. The rules have not changed a great deal since the days of the first shell. Shells take the line of input from the user and break it into "words" that are separated by spaces or tabs. The first word on the line is taken to be the name of the command, the remaining words are each separate arguments that are passed into the command. So, when we type

```
$ cp one two
```

the `cp` word is the command name, and the shell will locate the command file that goes with the name. The remaining strings are passed into the `cp` command as arguments; the zeroth argument will be `cp`, the first argument will be the

string one, and two will be the second argument. The cp command interprets the first argument as the source file for copying and the second as the name of a destination to which to copy the data.

Of course, the shell can do more than just take our input and process it. The earliest form of additional processing provided by the shell was shell *globbing*, the expansion of stars and question marks in file names. It gained its name from the file /etc/glob, which was the discrete program used as a shell "helper" in early systems. When the shell encountered a star or question mark, it ran /etc/glob to do the work of expanding file names.

It's important to understand that the globbing function happens *before* the exec system call is made. For example, when you type

```
$ echo a*
```

the shell looks for all the files starting with "a" in the current directory, sorts the resulting list into alphabetic order and passes the complete list of file names as parameters into the exec call. Expanding the names in this way means that file name expansion doesn't need to be coded into every command, it exists only once in the shell. However, this method can sometimes have confusing results if you are not clear when it happens as the process is created.

Similarly, I/O redirection happens in the child code after the fork() and before the exec(). When we type

```
$ echo a* > out
```

the shell arranges that the echo command is run with it's standard output set to the file out. Setting up the new output channel happens after the fork and before the exec. A side effect of this mechanism is that the destination out file is created *before* the command is executed. This can sometimes be counter-intuitive. An attempt to add the contents of one file to the end of another might reasonably be written as:

```
$ cat a b > a
```

but this fails (usually horribly) because the shell will open a new file called a *before* the cat command is executed. Opening a new file with the same name as one that exists results in truncating the file to zero length. The cat command will now copy a zero-length file a and the contents of b to a. Bill Joy implemented an option (the noclobber option) in csh to prevent the I/O redirection option from destroying an extant file, so I guess something horrible happened to him at some point.

Quoting

Using white space to separate the words on a command line is convenient largely because the space bar is a large friendly area at the bottom of the keyboard. In fact, in most shells, the word-separation characters are specified in a shell variable, allowing the user to change the character set should they wish. However, most people stick to the default.

Generally, the use of white space as a separator has meant that, on UNIX, we don't use file names that contain embedded spaces. You are at liberty to create a file with any name you wish, and you are able to create files with embedded spaces, however, to handle them with the extant shells you need to know how to include a space in the middle of an argument string for a command.

As this article has progressed, we've also begun to build up a list of special characters (meta-characters) that the shell uses for its own purposes: *, ?, &, <, > and so on. There are more. For instance, I've not mentioned shell variables that are invoked by placing a dollar symbol before the variable name. For example, for Bourne shell and derivatives

```
$ DEST=/usr/share/man
$ ls $DEST
```

or for csh and derivatives

```
% set DEST = /usr/share/man
% ls $DEST
```

In both cases, the shell expands the variable "in place" to be its contents before the ls command is executed. The command that's run is

```
ls /usr/share/man
```

It's clear that a method of quoting is required to allow us to pass all the characters that form part of the shell's syntax into commands, while stopping the shell from doing what it normally does with the characters. There are, of course, several methods of doing just this.

UNIX uses the backslash character in many applications to be an escape character. It usually means "take the next character literally, don't treat it as a special character." All shells permit the use of backslash in this manner. For example,

```
$ echo \$DEST \> fred
```

will print

```
$DEST > fred
```

Things begin to get more exciting when you are using a command that uses meta-characters. Let's try and find the values for the dollar symbol from /usr/pub/ascii using grep. Our first attempt might be

```
$ grep $ /usr/pub/ascii
```

The shell will leave a single dollar character alone and will pass it into the grep command unchanged. However, if you try this, you'll find that it lists all the lines in the file. (Pause here and see if you know why, before reading on.)

The argument to grep is a regular expression, and the

dollar sign is one of the meta-characters used in a regular expression match: $ matches the end of the line. By definition, each line in the file has an end-of-line character, so this regular expression matches all the lines. The single $ argument won't do what we want, we need to use backslash to get a dollar into the program:

```
$ grep \$ /usr/pub/ascii
```

This again lists all the lines in the file. Why? Well, when the shell sees a backslash it reads the next character and discards the backslash, so this command is essentially equivalent to the first attempt. We need to get a backslash and a dollar sign into grep:

```
$ grep \\$ /usr/pub/ascii
```

The shell passes \$ as the match expression into grep and the command will find all the lines containing a dollar symbol.

Using backslash like this can get tedious, especially in long strings. It would be better to have a way of quoting chunks of text without having to worry about what it contained. Shells provide two ways of doing this. First, you can enclose some text in single quotes:

```
$ echo '$DEST > fred'
```

The output from this is the same as the previous echo example, however, the source is clearer and less cluttered. The contents quoted by the string will be passed intact through to the command as a single argument after the quotes have been removed. The only character that cannot appear inside a single-quoted string is a single quote. However, quoting in shells is an area of great divergence, different shells implement things in different ways. For example, consider the following command:

```
/bin/echo 'a\\b'
```

I'm using /bin/echo to avoid the use of any shell built-in echo function. Because single quotes are supposed to pass things through unchanged, then you might expect to see this print a\\b. However, a little experimentation with the shells on my machine produces the following:

```
/bin/sh:    a\b
/bin/ksh:   a\b
/bin/csh:   a\b
/bin/bash:  a\\b
```

I am unsure whether bash is wrong. It has probably done the right thing, rather than simply following what the Bourne shell did. In all the shells, the escape character is actually not useful inside single quotes. A backslash cannot be used to insert a single quote. On balance, I'd prefer to see the shells leave my double backslash alone.

There are occasions where you would like to pass a single argument to a command, perhaps including spaces, but have the benefit of variable substitution. Using double quotes around a string achieves this, so

```
$ echo "$DEST > fred"
```

will print

```
/usr/share/man > fred
```

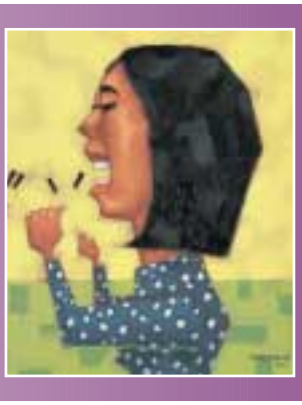
assuming that the content of DEST is unchanged from the previous setting above. You can use double quotes to force the quoting of a single quote:

```
$ echo "It's a single quote"
```

Another useful trick is to realize that shells are text-processing languages and will perform string concatenation for you. I often use combinations of quotes to ensure that complex and lengthy strings are left alone by the shell, except where I want things to happen. For example,

```
$ echo '$DEST = '"$DEST Doesn'"'t it?'
```

will generate one argument to the echo command. Combinations can become indecipherable if you're not careful. Most of these tricks are only needed when you are attempting to get complex statements involving meta-characters into some of the super tools like sed or awk.



Quoting in shells is an area of great divergence, different shells implement things in different ways.

When you are creating complex scripts for the super tools, then you should avoid using csh for your scripting language. One reason is that the Bourne shell and its derivatives handle newline characters in a much more flexible manner. To get a newline into a quoted section in the Bourne shell, you just include it:

```
echo 'Here is a
newline'
```

You'll find that csh is line-oriented and insists on a backslash before the newline character to achieve the same effect. Even then, things don't work too well. If I am writing awk scripts

for the Bourne shell, I'll often write things like the following:

```
awkprog='
        {sum = sum + $5}
END     { print sum}'
/bin/awk "$awkprog"
```

This lays out the `awk` program in a readable manner, and passes it into the command cleanly in a shell variable. Incidentally, the command is intended to be used to count the bytes in a directory:

```
$ ls -l | sh aw
```

where `aw` holds the script above. The `awk` command in the script processes data from the standard input channel of the script. If you attempt to do this in `csh`, then you need to add backslashes at the end of all the newlines in the single-quoted section:

```
set awkprog = '\
        {sum = sum + $5}\
END     { print sum}'
/bin/awk "$awkprog"
```

But the program blows up when the `awk` command is invoked. The sad fact is `csh` doesn't like embedded newlines in variable contents.

You'll also find that combinational quoting in `csh` is pretty broken. Again, things that you can do trivially in the Bourne shell just don't work in `csh`. For example,

```
echo "dollar \$"
```

works in `sh`, but not in `csh`.

I gave up writing scripts in `csh` aeons ago, for these and other reasons. If you want to learn to write scripts, use the Bourne shell. Your scripts should be portable to all the machines in the world. One caveat: some of the Bourne shell clones haven't implemented quotes in exactly the same way as the original program.

Further Reading

The UNIX Programming Environment, by Brian W. Kernighan and Rob Pike, is published by Prentice Hall Inc., 1984, ISBN 0-13937-681-X. More reasons to hate `csh` can be found in Tom Christiansen's `csh` article reproduced in *UNIX Power Tools,* by Jerry Peek, Tim O'Reilly and Mike Loukides, published by O'Reilly & Associates Inc, 1997, ISBN 1-56592-260-3.

***Peter Collinson** runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.