by Peter Collinson, **Hillside Systems**

CATHY GENDRON

Spaces

Last week, I was presented with a file hierarchy on a ZIP disk. The files were mostly images of products to be displayed on the Web. The person who created the hierarchy named each image file with a long name that described the file's contents. The file names were long descriptive phrases and because he was using Windows, he used spaces to separate the words in the phrase.

Now, on the whole, we don't use spaces in file names on UNIX. Actually, there's nothing to stop you creating a filename that contains any character. UNIX has always done what it's been told and rarely complains when the user tries to do unwise things. Filenames can contain any character, including non-printing ones like Control-C. As far as the UNIX kernel is concerned, there is nothing special about a file name. It's just a sequence of characters that's the entry in a directory that maps onto an inode number that's a file on the disk.

Over time, we have adopted a bunch of ad hoc conventions about file names that helps various programs to operate. For example, the C compiler expects its source files to end in `.c`, and will automatically compile intermediate object files that end in `.o`. If you use MIME mail to send files, then the file suffix is used by the MIME processor to deal with the file appropriately. So, a file ending in `.jpg` will be treated as an image file in JPEG format. In recent years, we've tended to use three letter suffixes as a sop to Windows. You will find that a file ending with `.jpeg` should also be treated as a JPEG file. However, conventions about file suffixes are not really UNIX constraints. They are helping the operation of programs that run on UNIX.

Of course, some file names are inconvenient. You can create a file called `-` perhaps by writing it from an editor. Problems then ensue, because when you use this file as a program argument, the file name will then look like an option to the program and not a file name. Trying to remove this file:

```
$ rm -
```

yields

```
usage: rm [-fiRr] file ...
```

For some time, the `rm` command has treated `-` on its own as a special option, meaning "don't process any more options," so

```
$ rm - -
```

will delete the file called `-` (or any file that starts with a `-`). This is special behavior for the `rm` command. Actually, all programs should follow the standard way of stopping processing any further options, namely using `--`, so

```
$ rm -- -
```

will also remove the file called `-`, and you will be able to use `--` in any command to handle the file called `-`. However, this is a pain, and avoiding filenames that start with a `-` is a prudent thing to do.

Spaces

Spaces in filenames are similarly inconvenient, because shells expect to split the command line into "words" using a space as a separator. Shells will treat several contiguous spaces as a single separator and will also treat tab characters as separators, because there's no way for a human to look at the screen and tell the difference between a tab or some number of spaces.

We can deal with filenames with spaces by using the quoting facilities of the shell:

```
$ mv 'with space' no_space
```

or

```
$ mv "with space" no_space
```

either form is used to avoid the normal interpretation of spaces. But filenames with spaces can become a problem when we try to deal with whole file hierarchies. Let's say we want to find something in all of the files. An initial stab may use the backquote operator:

```
grep something `find . -type f`
```

The `find` command discovers all the regular files in a tree and prints the list of files on its standard output. This list is used as a set of filename arguments to the `grep` command. The command sequence can sometimes fail if the output from the `find` command is immense, because some systems have a limit on the number of bytes that can be present in the argument list. POSIX only requires that the system supports a maximum argument list of 4,096 bytes. Solaris 8 allows 2,096,640 bytes for 64 bit programs and 1,048,320 bytes for 32 bit programs.

Also, the command will fail if any of the file names contain spaces, because the output from the `find` command is subject to the normal shell parsing rules that are applied when the output from the `find` command is used as the argument to `grep`. So if `find` discovers our file `with space`, it will be presented to `grep` as two arguments, and `grep` will fail to find a file called `with` and a file called `space`.

Using a Loop

The backquote approach doesn't work when we have spaces in filenames, but can we use another approach? A shell loop might do the trick. I am using shell loops for Bourne Shell and derivatives, the code doesn't work in `csh` or its children. A loop will also allow us to execute a sequence of commands and this can be useful. The basic loop pattern is

```
$ find . -type f |
  while read name
  do
    grep something $name
  done
```

The output from the `find` command is piped into the loop controlled by the `read` statement. The contents of each line in the list are placed in the `name` variable, and the code in the loop is executed. Here I'm using the `name` as an argument to the `grep` command, so the command is run on every file found. If the file contains `something` then any matched line in the file is printed.

There's a slight wrinkle here, sent to me by a reader a long time ago. If the `grep` command is given a list of files to search, then it will print the filename in addition to any matched line. This is great, because you can easily identify the files you are looking for by inspecting the output. If `grep` is only given one filename, then it assumes that you know the filename, and will only print matched lines, but no name. So, in a loop like above, some way is needed to give `grep` another filename argument. If you change the `grep` command above to read

```
grep something $name /dev/null
```

then this does the trick. The `grep` command is given two names and will print the filename and matched lines if the file in `$name` contains the needed information. Of course, `/dev/null` never contains the data, so it's a safe additional argument.

Well, the script looks reasonably sound now. What happens? Each line output from the `find` command is a filename and will end up in the `$name` variable. The `grep` command is run on the file. If the file contains the desired string then `grep` will output the matched line preceded by the filename. However, the command still doesn't deal with embedded spaces in filenames coming from `find`. If a space appears in the name, then the shell will pass `$name` into `grep` as two arguments. However, we can get around this by quoting the variable:

```
grep something "$name" /dev/null
```

and we are in business. Shell variables are expanded inside double quotes, so the `grep` command is presented with a filename argument taken from the `$name` variable.

Renaming the Files

However, having to quote variables is a pain. I'd prefer not to have spaces in the filenames. I might want to use the backquoted `find` technique I used at the top of the article on the complete file system tree. To avoid problems, I usually change spaces in filenames to underscores, which preserves the original intention of the naming scheme, presenting a file name with several words in a readable phrase. This name change operation is likely to be crucial, so let's write a shell function that does the job.

```
chgname() {
    echo "$1" | sed -e 's/[ ][ ]*/ /g
        s/[ ]/_/g'
}
```

A shell function allows you to group a set of commands into a block and execute the commands by calling the function name, as if the function name was a regular command name. Of course, commands can have arguments, and shell functions are the same. Inside the function, arguments are accessed by the *positional parameters*, `$1`, `$2` etc. You may recall this syntax is also used to access the parameters in shell scripts. In fact, shell functions behave identically to shell scripts, except that you don't use the `exit` command to force the function terminate, you use `return`. It's often possible to take a complete shell command file and make it into a function in the script you are developing. I'll often develop a function in a small separate shell script before using it in the work in progress.

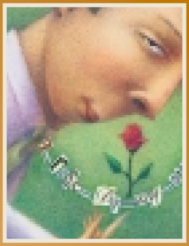
Incidentally, `ksh` and derivatives have some new syntax to define a function, so you can say

```
function chgname {
```

I tend to stick to the older form, because it's more portable.

What does `chgname` do? It's easy to see that the first argument to the function is passed into `echo` and used to create an output pipeline of a single line. The output is passed into the `sed` command that will output a result on the standard output.

The `sed` command makes two sets of changes to the data. First, it replaces groups of multiple spaces by a single space using the substitute command. The `g` (for *global*) at the end of the replacement makes the change happen everywhere on the line.



It's often possible to take a complete shell command file and make it into a function in the script you are developing.

There *is* a little magic happening here, but the regular expression idioms being used are common. First, to make the regular expression more readable, I put any space that is to be matched in alternation braces as a way of highlighting that I want to match a single space. It's not necessary, but does tell you instantly what is going on with no confusion.

If you are trying to match one or more spaces, then it's a common mistake to write

```
[ ]*
```

forgetting that the star operator matches zero or more occurrences of the previous character. So `[ ]*` matches any character on the line, because there are zero occurrences of any spaces. The action is somewhat counter-intuitive. It inserts spaces between every character on the line in what seems to be a mysterious way.

What we want to match is a space, followed by zero or more spaces, and we do this with a space `[ ]`, followed by a space repeated zero or more times `[ ]*`. Actually, most regular expression matching systems now use a plus symbol to mean *one* or more matches–so, for example, in Perl, we can say:

```
[ ]+
```

and have the regular expression behave in the same way as the older syntax:

```
[ ][ ]*
```

Having changed the repeated spaces to a single space, the next `sed` command in the script changes all spaces on the line to an underscore. Finally, `sed` will output the amended line to the standard output channel.

Making a Script

Having created a function that will change file names, we can use it to make a small script that scans the file hierarchy and use the `mv` command to rename files from their old name with spaces to their new name with underscores.

We'll use the `find` command to seek out all the files we need to alter, pipe the output into a loop, and perform the necessary renaming function. It's not quite that simple–things never are. Consider a little heirarchy where we have a directory called

```
d space
```

that contains two files called

```
f space 1
f space 2
```

The `find` command of

```
find . -name '* *'
```

which looks for all files that start with *something*, have an embedded space, and then finish with *something*. The name match here is the same as the shells, so the star will match zero or more characters. When applied inside our test file hierarchy, the command will generate something like:

```
./d space
./d space/f space 1
./d space/f space 2
```

The list is already sorted by the way the files are laid out on the disk. By default, `find` prints the directory name, and then descends into that directory to investigate its contents. However, I certainly would pipe the output of `find` through `sort` to ensure that the file order I want will be guaranteed. I want to make sure that the directories appear in the list before any files that exist in the directory. By passing the list through `sort`, we guarantee the order of directories and files, simply due to the length of the string that's being used.

Now, let us work through what we plan to do in the loop.

The first time round the loop, we will pass the first line from find through chgname and generate a mv command that does:

```
mv "./d space" ./d_space
```

This is looking hopeful–it does what we want to rename the directory. What happens on the second line from find? Remember that we're planning to pass the line into chgname and use its output in the mv command:

```
mv "./d space/f space 1" ./d_space/f_space_1
```

Sadly–and I hope you saw what was coming–this no workee.

By now, we've already renamed the directory d space to d_space. The pathname ./d space/f space 1 no longer exists and we need to use the command:

```
mv "./d_space/f space 1" ./d_space/f_space_1
```

So, the script needs to be a little more intelligent about paths in the hierarchy. By sorting the list into "string" order, we've ensured that directories will be renamed before the files they contain, but the arguments to the mv command need to be constructed somewhat more carefully.

Splitting Pathnames

There are a pair of basic tools that can be used to split UNIX pathnames into their constituents. The command basename has been kicking around UNIX systems since the earliest one that I used. The job of basename is to split off the file part of the name and print it on its standard output:

```
$ basename './d space/f space 1'
```

will print

```
f space 1
```

The complementary command, dirname, prints the other half of the path:

```
$ dirname './d space/f space 1'
```

will print

```
./d space
```

For the record, basename also understands suffixes, and can be used to remove them:

```
$ basename fred.txt .txt
```

will print

```
fred
```

The two commands, basename and dirname, give you the control to manipulate filenames to derive new ones from old in a programmed way. We can use them to extract the portions of the filename that find generates. We can now create the script.

```
#!/bin/sh

# insert code for chgname

find . -name '* *' | sort |
while read name
do
    file=`basename "$name"`
    stem=`dirname "$name"`
    nfile=`chgname "$file"`
    nstem=`chgname "$stem"`
    if [ "$file" != "$nfile" ]
    then
        mv "$nstem/$file" $nstem/$nfile
    fi
done
```

As discussed above, we use find and sort to generate a list of files to be processed in directory and then filename order. The list is dealt with one file at a time by the loop. For each file, we split the name to be processed into its filename and stem using the basename and dirname commands. Note that I'm being careful about quoting the argument in $name and anything derived from that because it may contain one or more spaces. After using chgname to alter the filename and stem into their new forms, we test whether the new file name differs from the old and issue the mv command to rename the file if it is needed.

The result is a working script that renames any file or directory that contains a space into one that contains an underscore.

Changing the Parsing Rules

I've had to be especially careful about quoting the output from the find command in the scripts above. This is necessary, because the shell wants to split its input into words using space, tab or newline as a separator. However, you can change the set of characters that the shell uses to parse input lines by setting new values into the IFS variable. Adding:

```
IFS='
'
```

just before the loop alters the set of characters the shell uses to parse input lines. So, in the example above, I am setting IFS to just contain the newline character, removing the special meaning of space and tab. If we place this statement just before the loop, then we change the behavior of the read statement and also any variable insertions into the command line. We can then stop worrying about space being a "special" character. Actually, we don't really need this ability in the example within this article.

However, the ability to change the separation character set can be used to allow the shell to process files that use different separators. The classical case is processing the password file where the colon character is used as a separator.

```
IFS=":"$IFS
cat /etc/passwd |
while read name restofline
do
       echo $name
done
```

allows us to process the password file by picking off all the names which happen to be the first field of the password file. The first line prepends the colon character to the extant `IFS` set and affects the `read` statement which parses the input line from the password file into two chunks. The name is placed into the `name` variable, and the remaining information ends up in the `restofline` variable.

Incidentally, the IFS variable has been used several times to attack and subvert shell scripts. The attacks usually center around definitions like:

```
CP=/bin/cp
...
$CP old new
```

This looks very innocuous. However, because the IFS variable is in the environment, we can set it to something odd before we execute the script and obtain an interesting result. If we call the script adding the slash character to the `IFS` string, for example, then the `CP` variable will be defined as:

```
bin cp
```

and the `$CP` line will be executed as:

```
bin cp old new
```

We have suddenly introduced a new command into the script–one that the author didn't intend to be there–called `bin`. If the shell script is running as root, I can define a private program called `bin` that is actually a shell program, and with luck, I will have an interactive shell running with super-user privileges. The solutions are twofold: first, always place your definitions in quotes so they can't be subverted by the `IFS` variable; second, always define a `PATH` variable that specifies a known set of search directories in publicly available scripts. I've also seen some scripts that explicitly set their IFS variable to ensure that what they are processing is what the author intended.

***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`.