by Peter Collinson, Hillside Systems

DENISE ORTAKALES

Shell Control Files

My article on shells ("A Shell Road Map," December 1997, Page 26) attracted quite a bit of email, showing that shells are still an important issue for some users, even in this pointy-clicky draggy-droppy age. Of course, the email started me thinking about what was missing from that article, so this month, I'll look at how shells are set up and tailored for your personal use.

When you are given a login on a UNIX machine, the hopefully friendly systems administrator will allocate the shell that you will use as your login shell. Actually, this is not quite the whole story, because all shells have start-up files (usually starting with a "dot") that lurk around your home directory. You probably know by now that the UNIX `ls` command suppresses file names starting with dot, and that many programs use this fact to place initialization files "invisibly" in your home directory. Actually, the dot files seem to breed like rabbits. I have about 60 files starting with a dot in my home directory today, of varying antiquity.

Anyway, you'll trot along to get your UNIX login and very probably be given the pet shell of the systems administrator along with his own start-up files, hopefully suitably amended for your use. It may be that there is some standard setup for your site, and you'll be given that. Initially, it's probably a good idea to ignore these start-up files. They will contain various parameter settings, file names and paths that will make your system work for you. However, later you may want to tailor things, so a knowledge of what files are used for what may be useful.

Different shells handle the start-up files in different ways, and the situation with recent shells is often complex because they have tried to maintain some semblance of backward compatibility. I always think that knowing how something has developed helps you untangle what is going on. So let's start with the Bourne shell.

The Bourne Shell

The Bourne shell was designed to run on UNIX Version 7, and that system supported the process "environment," which was a new idea for passing information between processes. Whenever a new process is started, it inherits a set of environment strings of the form `NAME=value` from its parent. The strings are cloned from the parent process on the `fork` operation that creates the new child process. They survive the `exec` system call which makes a running process transmute into a new program that may do something different. Incidentally, you can see the strings that are set for your shell by typing `set` into your shell if it derives from the Bourne shell (`sh`, `ksh` or `bash`) or `printenv` into a C shell variation (`csh` or `tcsh`).

Various programs use different environment strings. Some environment variables are used directly by the shell. For example, the `PATH` variable contains a colon-separated set of directories that

are searched by the shell whenever a command is typed. Some are used by application programs. For example, the EDITOR variable can be set to the name of your favorite editor, and this program is used by many commands that wish to start an editor on some text file.

Any process can add new environment strings, delete strings or change their value, and all of its children will inherit the new values. When you log into the system, the login process will establish some environment strings that are passed into the shell before it starts. A quick look at my Sun's login program shows that it sets SHELL to the name of your shell, HOME to your home directory, LOGNAME to your login name, MAIL to the default mail path and PATH to the default command search path.

However, it is convenient for you to be able to establish your own set of environment variables, in order to change the default PATH setting, for example. You can do this by typing commands into your shell, but this is tedious; it's much more convenient to take the "usual" setup from a control file. With the Bourne shell, the control file is called .profile and is stored in your home directory.

When the Bourne shell is started as part of the login sequence, it will look for a .profile file in your home directory and will read commands from it. This only happens when the shell is being started as part of the login sequence, and there's a small piece of magic that tells it that.

Interactive Login

It's the job of the login program to ask the user for a password, check it and permit the user to log in if the password check succeeds. Permitting the user to log in is simply a matter of running a shell, and the login program starts the shell by using the exec system call to overlay itself with the shell of choice for that user.

The exec system call establishes a set of strings that are the parameters to the command that it is invoking. The first parameter is the name of the command, usually the name of the file in which the command is stored. So if an exec system call is running a command such as

```
$ ls -l /bin
```

the exec system call that will be invoked will look something like

```
exec("/bin/ls", "ls", "-l", "/bin", NULL);
```

The first parameter is the address of the ls command in the file system; the second is the name of the command that is being invoked; all the subsequent parameters are taken from the command line; and the list is terminated by the magic symbol NULL, which happens to be the number zero.

When the new ls process begins to run, it is presented with the arguments (except for the file name), decodes them and uses the information to make things happen the way the

user has asked. The command can also see the name by which it was called. Some programmers have made use of the ability to see the name of the command. A good example is the `vi` editor family. The family consists of the regular `vi` editor, a restricted version that will not write files called `view`, an editor for beginners called `vedit`, a line-based editor called `ex`, and finally, a simplified version of `ex` called `edit`. If you look on your system, you'll find that these names appear in `/bin`, but they are all links to the same file. The program looks at the name by which this is called and changes its behavior accordingly.

To summarize this small diversion: A process that calls the `exec` system call can set up the name of the program it is calling, and the program that is called can see and decode that name. As we have seen, shells will always pass the name of the file in which the command resides into the command as its name. However, when the `login` process `exec`s to the shell, it will set the first character of the command name to a hyphen. The shell will see this and will know that it's been called from the `login` process, and uses this information to run commands from a start-up file. For the Bourne shell, it will look for the `.profile` file on your home directory and read commands from there.

How Start-Up Files Are Run

It's a good idea to understand a further nuance about the way all this works. Let's say you place some commands into a file, make the file executable and type that file name as a command. Your shell will run that new command file by starting a new shell to read the commands and execute them. Any environment-setting statements in that command file will not affect the shell to which you are talking, because they are being run in a subshell.

However, we actually want to set up strings in our shell, so the contents of the `.profile` file are executed directly by the login shell. Most shells have syntax that allows you to do this "by hand." There's a special command (dot) with the Bourne shell,

```
. file
```

that tells the shell to run the commands *in the current shell*.

Start-up files are run automatically in the current shell using the same mechanism. This is no big deal if the commands only set environment variables. However, the `.profile` file is often used to run other commands, some that establish the way you want to work. For example, I have

```
stty cs8 -istrip -parenb erase ^H
```

in my start-up files, so when I log in, my keyboard is sending 8-bit characters, and my default erase character is set to backspace. Some people also place other commands in their `.profile` file, for example, `uptime` or `who`.

However, if any of these commands fail for some reason,

then you can be "stuck," unable to log in. If you are messing about with login start-up files, then always make sure that the new files work before logging off completely. Using the "dot" command is perhaps one way of checking the files, but you should always try to log in with the new setup before you completely log off the system. On a workstation, you can simply start a new window and use `telnet` or `rlogin` to log into your machine again:

```
$ telnet localhost
```

This will check that the changes you have made work properly.

Actually, on most modern systems, the Bourne shell will also look for commands in `/etc/profile`, and then for your own `.profile`. This allows the systems administrator to establish settings for your system and for you to inherit the variable values. For this reason, it's always a good idea to retain previous settings of certain variables. For example, if you are adding a private `bin` directory to your `PATH`, type

```
PATH=:/home/pc/bin:$PATH
export PATH
```

rather than setting up a complete new `PATH` based on what you find in that variable today.

The C Shell

When the C shell appeared, it extended how start-up files are handled. The shell supports two start-up files: `.login`, which is invoked when you log in, and `.cshrc`, which is invoked whenever a `csh` is started. It also supports a file that's called when you log out, `.logout`. Some people use this file to clean up temporary files when they leave the system. In the days of terminals, its greatest use was to run the `fortune` program to generate a pithy epithet to leave on the screen of an unused terminal.

The designers of `csh` decided to have two start-up files because they had added the notion of aliasing to the shell. A user can type

```
alias rm 'rm -i'
```

so that whenever the user types `rm`, the interactive flag is automatically added to the command, ensuring that you are asked to confirm every file deletion. Aliasing allows you to create a private command set without having to establish a private command file, and this can be useful.

However, there's no easy way of passing aliasing definitions from one shell to another. You can see this by typing

```
% alias hello 'echo hello world'
% hello
hello world
% csh
% hello
hello: Command not found
%
```

The first line establishes an alias to `hello`, which we test by calling it. Then we start a new `csh` and find that the alias has disappeared because it only exists in the login shell. The example is perhaps artificial, but the second `csh` could be invoked from inside another command, perhaps an editor. When you start a new shell, you would expect all your aliases to be established and would be surprised that suddenly your command set had changed back to what it was previously. So, if aliases are to be useful, we want them to be established automatically in every shell that we run. To achieve this, `.cshrc` is invoked every time the shell is started.

When an interactive `csh` is started at login time, the shell will execute commands from `/etc/.login`, then commands from `.cshrc`, and then commands from your own `.login` file. The original idea is that you should put your terminal and environment setting commands into the `.login` file and put any command that you want to be executed on the invocation of every shell into your `.cshrc` file.

The X Window System and widespread use of `rlogin` have caused many setup values to move into the `.cshrc` file because when you start a new window, you'll start a new shell. But it won't be a login shell, so only the `.cshrc` file will be executed when the window is opened. Many interactive control features now need to be in your `.cshrc` file so they are invoked every time you start a shell. For example, `.cshrc` should include your `history` setting command so that shell history is functional, and could include

```
set filec
```

if you want to use file name completion in `csh`. If you turn it on, you'll find that you can make the shell automatically complete file name arguments to commands for you. You type the first few letters of a file name and hit the escape key. The shell will read the directory and match the string that you have typed with the names of the files that it finds. If the name is unambiguous, the shell will supply the remaining characters. If the name isn't unique, the shell will beep you to enter some more characters to ensure a match.

You can also set the shell variable `fignore` to tell `csh` what file extensions to ignore in its file completion. For example, people commonly set this to `.o` so that object files from compiles are ignored. If, however, the only possible completion includes a suffix in the list, it is not ignored.

It's often a good idea to recognize that some of the commands in `.cshrc` are only aimed at supporting interactive use. You can easily code in a test that looks for the presence of the `prompt` variable and use this test to discriminate whether the shell is interactive or not.

```
# We start with commands that are
# always needed
set path = (...)
# the csh path variable is 'written
# through' to the PATH environment
# variable
#
```

```
# Now test for the presence of the
# prompt variable
if ($?prompt) then
# we have a prompt - put interactive
# set up commands here

   set history = 100
   set filec
   alias rm 'rm -i'
# etc etc

# end of interactive set up

endif
```

Incidentally, rather than enclosing the interactive statements in an `if` statement block, some people put

```
if (!$?prompt) exit
```

at the top of the `.cshrc` file. The *UNIX Power Tools* book (see below) points out that the action of the `exit` statement is unpredictable in different implementations of `csh`. In some cases, you'll find that you will be logged out because you've told your login shell to exit. So you should use the nested `if` structure or a `goto` jumping to a label at the end of the file.

The version of `csh` that supports line editing and command completion, `tcsh`, is designed to be compatible with the `csh` that is resident on your machine. It will read the standard `csh` systemwide files that are present on your machine. You can supply a file called `.tcshrc` that supplants `.cshrc` so you are able to add specific `tcsh` setup commands.

The Korn Shell

The Korn shell started life as a modified Bourne shell and will automatically read your `.profile` file when you log in. It will also read `/etc/profile` to permit systemwide initialization. Unlike the Bourne shell, the Korn shell provides for a file to be read whenever the shell is started. There is no well-known, default name for this file; instead the name is taken from the contents of the `ENV` environment variable. People often name the file `.kshrc`, because this name fits in with `.cshrc` and other shell start-up files.

You set the `ENV` variable in your `.profile` and ensure that it is exported to the environment. Then whenever a new shell is started, it will find that `ENV` is defined and will run the commands that it finds there. The contents of the file will be executed after the `.profile` file has been read, assuming that file sets `ENV`.

Again, these days, people find that most of their setup has migrated into the `.kshrc` file and need to find a way of distinguishing between interactive and noninteractive start-up. This can be done by checking whether the interactive (`-i`) flag is set. The Korn shell maintains a variable accessed by `$-` that contains a list of all the flags that are set. So the best way to find whether a shell is interactive is

```
case $- in
*i*)
# interactive setup commands
    ;;
esac
```

The `*i*` in the case option will be matched if the `$-` variable contains the character `i` and the commands will be executed.

Incidentally, `ksh`, `tcsh` and `bash` will only read commands from files that are owned by you, so it's harder for someone to trick you into starting a shell and then inadvertently running some commands that they have established to compromise the security on your system.

The Bourne Again Shell

GNU's Bourne Again shell (`bash`) permits several configuration options for start-up files. It's a bit of a cross between `csh` and `ksh`. For login shells, it reads systemwide initialization from `/etc/profile` and then looks for `.bash_profile`, `.bash_login` and `.profile`. It will execute the first one of these files that it finds.

If `bash` is started interactively, which it knows from looking at whether or not it's talking to a terminal, then it will look for start-up commands in `.bashrc` in your home directory. Finally, when invoked from a script noninteractively, `bash` will execute commands from a file whose name it finds in `BASH_ENV`.

All this seems complex to me, and so my `.bash_profile` contains the single line:

```
. ~/.bashrc
```

which reads commands from my `.bashrc` file so all my setup commands are placed in that file. Because `bash` is compatible with `ksh`, it supports the `$-` variable and so you can use the same interactive check that was described above for `ksh`.

Actually, for some considerable time, I've maintained a file called `.set_shell_vars` that contains a set of Bourne shell environment setting commands. I place a "dot" command in the start-up files for `sh`, `ksh` and `bash` that reads commands from this file to establish my basic environment settings. These include values for `PATH`, `EDITOR`, `PAGER` (I prefer to use `less` rather than `more`), `MANPATH` and character set (I set `LC_CTYPE=iso_8859_1`) so that I can type Latin-1 accented characters easily.

Finally

UNIX Power Tools by Jerry Peek, Tim O'Reilly and Mike Loukides, published by O'Reilly and Associates Inc., ISBN 1-56592-260-3, is in its second edition and is a great source for information on the issues covered by this article.

Thanks to Garry J. Garrett and John Caruso for the email input that started me thinking about this article.

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 a SPARCstation 2. Email:* `pc@cpg.com`.