5 Job Control

The UNIX operating system accommodates the execution of programs in two modes: foreground mode, where the terminal or window being used does not prompt for a response until the program is completed; and background mode, which executes the program without tying up the terminal or window from which the command invoking the program was executed. By default, the commands you type are carried out in the foreground. In an environment with multiple windows, these distinctions are not so critical, since additional windows can be opened if the one in which a command was entered is not available. When using a terminal with just one window, it can be very useful to run one or more commands in the background while an interactive session is being carried out. Even in a windowed environment, it can often be useful to run a program in the background if you know you will be logging off before the program has completed executing, since jobs in the foreground will terminate when you logoff. To make it known the the UNIX operating system that you wish to run a job in the background, terminate the command with an ampersand (`&`). (Under some versions of the Bourne and Korn shells, you may need to use the `nohup` commands if you want to make sure that a background job will not terminate when you log off.) When running a job in the background, output from the background process will still be displayed on your computer screen, so it's usually a very good idea to use redirection (described in the first part of this article) to redirect both the standard output *and* the error output to a file. This is especially important if you will be logging out before the job is completed, because in that case there will be no way to retrieve output which has not been redirected. Remember that the ampersand should be the very last character of the command you wish to place in the background.

When a job is running in the foreground, you can communicate with it by sending a signal to it. The most commonly used signal is an interrupt signal, which is sent when you hit control-C on the keyboard (that is, type a "c" while holding down the control key). This will stop many programs, but some programs (notably many interactive statistical programs) will trap the interrupt signal and, for example, terminate the currently executing command and return to the program's prompt.

A more powerful signal, known as the quit signal, can be sent to an executing program using control-\. Most programs will be terminated after receiving the quit signal, but sometimes you will need to use the UNIX `kill` command if the program does not respond to a quit signal or if it is running in the background. For foreground jobs, you will also need to find the process id (PID) of the job, which can be obtained through the use of the UNIX `ps` command. For background jobs, you can use the job number (printed in square brackets when a job is put in the background or available through the `jobs` UNIX command) preceded by a percent sign (`%`) in place of the PID. You can find out more about these commands by accessing the online documentation through the UNIX `man` command.

An additional form of job control, available in most shells except the Bourne shell, allows you to temporarily suspend a running job. This can be very useful if you are running a job in the foreground, but then decide to put it in the background. To do this in a shell which supports it, you first suspend the running job by sending it a suspend signal, control-Z. Then you can place the job in the background with the UNIX command `bg`. The shell will respond with a job number, just as if the program was initiated in background mode, and the terminal or window will again be available for interactive commands. Remember that a job suspended with control-Z does not go away - it still uses some computer resources, and is available for restarting, either in the background (with the `bg`) command or in the foreground (with the `fg`) command.

6 Aliases and Startup Files

An alias is a way of abbreviating or redefining a command to customize it to your personal preferences. (Aliases are not available in the Bourne shell.) One very common example of an alias involves the UNIX `rm` command. By default, `rm` silently removes files without asking for your approval. (Any user who has accidently typed `rm *` knows that this default may not be very desirable.) There is an option to `rm`, namely `-i` which instructs `rm` to prompt you with the name of each file it would otherwise remove by default, giving you a chance to confirm that you actually want it deleted. To make sure that every time you use the `rm` command it is invoked with the `-i` option, you could issue the following command to the shell:

```
alias rm rm -i        # in c-shell and tcsh
alias rm="rm -i"      # in Korn shell
```

If you wish to override the meaning of an alias, and have it revert back to its usual default, precede the alias name with a backslash (\). So if you wanted to use `rm` without being prompted for verification of each file, you could use the command `\rm` instead of `rm`.

Naturally, you may not remember to type this every time you start up a shell, so each of the shells reads and executes a file of commands from your home directory every time they are started up. For the Bourne and Korn shells, the name of the file is `.profile`, while for `csh` and `tcsh`, the name of the file is `.cshrc`.

7 Shell Variables

Within each of the shells, you have an option of setting variables to values of your choice. These variables can be set locally, so that only the currently executing shell is aware of them, or they can be set so that they are transmitted to all new shells that are created from the current shell. In the Bourne and Korn shells, you can set a local variable with the syntax `variable=value`. Under the Bourne shell, you need to issue a separate `export` command to make the variable available to any newly created shells, while the Korn shell allows you to both set and export a variable using just a single `export` command with the syntax `export variable=value`. In `csh` and `tcsh`, a local variable is set using the syntax `set variable=value`, while global variables are set with the `setenv` command, omitting the equal sign. In all of the shells, a reference to a shell variable can be recognized by a dollar sign (`$`) at the beginning of the variable's name, although as you can see from the above description of how to set a variable, you do not use the dollar sign when defining the variables.

There are two main functions of shell variables. First, you can use a shell variable in a fashion similar to an alias, that is as an abbreviation for a longer or more complex piece of text. Unlike an alias, however, shell variables are not limited to just commands. For example, suppose you are accessing data sets in a directory with a complicated name, such as `/usr/data/survey/jan/rawdata`. If you entered the following command (or placed it in your `.cshrc` file):

```
survey=/usr/data/survey/jan/rawdata     # in Bourne and Korn shells
set survey=/usr/data/survey/jan/rawdata # in c-shell and tcsh
```

then you could refer to the file `/usr/data/survey/jan/rawdata/datafile` as simply `$survey/datafile`.

A more common use of shell variables is to communicate to other programs about the location of resources or personal preferences. For example, many programs will present output and invoke a UNIX pager to pause when the text has filled your screen. If there is a particular pager program you prefer, then you can inform these programs by setting the global variable `PAGER` to the name of the command you prefer. Of course there is no guarantee that all programs will look for this variable, but it is a fairly common practice. So you could set the `PAGER` variable to the value `less`, a popular public domain paging program with the commands

```
PAGER=less export PAGER          # in Bourne shell
export PAGER=less                # in Korn shell
setenv PAGER less                # in c-shell and tcsh
```

Note that under the Bourne shell, the command which sets the variable and the one which exports it to other shells can be typed on the same line. Other common variables which are used this way include PRINTER and EDITOR. In addition, the documentation of many UNIX programs includes information about shell variables specific to those programs. Most users find that setting these variables in their startup files (`.cshrc` or `.profile`) provides the most effective way of communicating their preferences to the programs they use.

8 History and Command Editing

One of the main motivations of developing new shells after the original Bourne shell was to add capability to "re-use" old commands. Much of the work which is done on a computer is repetitive, and it was felt that some mechanism to access and modify previously entered commands would be very useful. The C-shell (`csh`) provides this functionality through a history mechanism. First, the built-in command `history` will display recently entered commands, each proceeded by a number. (The number of commands displayed is controlled by the shell variable `history` which can be set in your `.cshrc` file.) In the C-shell, to access a previous command, you can enter its history number preceded by an exclamation point (`!`). In addition, the command `!!` repeats the last command, `!-1` repeats the next to last command,

and so on. There are also ways of modifying previous commands, which are explained in the UNIX manual pages for `csh`, most likely under the heading "History Substitution".

Rather than this form of access to previous commands, many users prefer to edit their previous commands using the editor of their choice, and then resubmit the commands to the shell. This functionality is built into the Korn shell (`ksh`) as well as the `tcsh` shell. These shells create a single line editing environment in which you can scroll your previous commands using the same key strokes as you would use to move up and down through a file, and to modify commands just as you would edit a line in a file. Upon hitting the return key, your edited command is submitted to the shell. These shells allow editing in either the style of `emacs` or `vi`, the two most popular UNIX editors. In the Korn shell you can choose between these two styles by setting an appropriate option with one of the following statements:

```
set -o emacs   # for emacs
set -o vi      # for vi
```

while in tcsh, you can use the `bind` command:

```
bind emacs     # for emacs
bind vi        # for vi
```

Once again, these statements should be placed in the appropriate startup file, so that the shell will always respond in the way you expect.