Shell

Whenever you login to a Linux system you are placed in a shell program. The shell's prompt is usually visible at the cursor's position on your screen. To get your work done, you enter commands at this prompt.

The shell is a command interpreter; it takes each command and passes it to the operating system kernel to be acted upon. It then displays the results of this operation on your screen.

Several shells are usually available on any UNIX system, each with its own strengths and weaknesses.

Different users may use different shells. Initially, your system administrator will supply a default shell, which can be overridden or changed. The most commonly available shells are:
Bourne shell (sh)

- C shell (csh)
- Korn shell (ksh)
- TC Shell (tcsh)
- Bourne Again Shell (bash)

Each shell also includes its own programming language. Command files, called "shell scripts" are used to accomplish a series of tasks.

Shell Script Execution

Once a shell script is created, there are several ways to execute it. However, before any Korn shell script be can executed, it must be assigned the proper permissions. The chmod command changes permissions on individual files, in this case giving execute permission to the simple file:

```
$ chmod +x simple
```

After that, you can execute the script by specifying the filename as an argument to the bash command:

```
$ bash simple
```

You can also execute scripts by just typing its name alone. However, for that method to work, the directory containing the script must be defined in your PATH variable. When looking at your .profile earlier in the course, you may have noticed that the PATH=$PATH:$HOME definition was already in place. This enables you to run scripts located in your home directory ($HOME) without using the ksh command. For instance, because of that pre-defined PATH variable, the simple script can be run from the command line like this:

```
$ simple
```

(For the purposes of this course, we'll simplify things by running all scripts by their script name only, not as an argument to the ksh command.)

You can also invoke the script from your current shell by opening a background subprocess - or subshell - where the actual command processing will occur. You won't see it running, but it will free up your existing shell so you can continue working. This is really only necessary when running long, processing-intensive scripts that would otherwise take over your current shell until they complete.

To run the script you created in the background, invoke it this way:

```
$ simple &
```

When the script completes, you'll see output similar to this in the current shell:

```
[1] - Done (127) simple
```

It is important to understand that Korn shell scripts run in a somewhat different way than they would in other shells. Specifically, variables defined in the Korn shell aren't understood outside of the defining - or parent - shell. They must be explicitly exported from the parent shell to work in a subsequent script or subshell. If you use the export or typeset -x commands to make the variable known outside the parent shell, any subshell will automatically inherit the values you defined for the parent shell.

For example, here's a script named lookmeup that does nothing more than print a line to standard output using the myaddress (defined as 123 Anystreet USA) variable:

```
$ cat lookmeup
print "I live at $myaddress"
```

If you open a new shell (using the ksh command) from the parent shell and run the script, you see that myaddress is undefined:

```
$ ksh
$ lookmeup
I live at

$
```

However, if you export myaddress from the parent shell:

```
$ exit
$ export myaddress
```

and then open a new shell and run the lookmeup script again, the variable is now defined:

```
$ ksh
$ lookmeup
I live at 123 Anystreet USA
```

To illustrate further how the parent shell takes processing precedence, let's change the value of myaddress in the subshell:

```
$ myaddress='Houston, Texas'

$ print $myaddress
Houston, Texas
```

Now, if you exit the new shell and go back to the parent shell and type the same command:

```
$ exit
$ print $myaddress
123 Anystreet USA
```

you see that the original value in the parent shell was not affected by what you did in the subshell.

A way to export variables automatically is to use the set -o allexport command. This command cannot export variables to the parent shell from a subshell, but can export variables created in the parent shell to all subshells created after the command is run. Likewise, it can automatically export variables created in subshells to new subshells created after running the command. set -o allexport is a handy command to place in your .kshrc file.

Shell Scripts - An Illustrated View

At the risk of sounding redundant, let's recap: shell scripts are simply files containing a list of commands to be executed in sequence. Now let's go a bit further and look at a shell script, line-by-line.

Any Korn shell script should contain this line at the very beginning:

```
#!/usr/bin/ksh
```

As you probably already know, the # sign marks anything that follows it on the line as a comment - anything coming after it won't be interpreted or processed as part of the script. But, when the # character is followed by a ! (commonly called "bang"), the meaning changes. The line above specifies that the Korn shell will be (or should be) executing the script. If nothing is specified, the system will attempt to execute the script using whatever its default shell type is, not necessarily a Korn shell. Since the Korn shell supports some commands that other shells do not, this can sometimes cause a problem. To be valid, this line must be on the very first line of the script.

Shell scripts are often used to automate day-to-day tasks. For example, a system administrator might use the following script, named diskuse here, to keep track of disk space usage:

```
#!/usr/bin/ksh
# diskuse
# Shows disk usage in blocks for /home
cd /var/log
cp disk.log disk.log.0
cd /home
```

```
du -sk * > /var/log/disk.log
cat /var/log/disk.log
```

Shown again - but this time with annotation - the script's processing steps are clear:

```
#!/usr/bin/ksh
# SCRIPT NAME: diskuse
# SCRIPT PURPOSE: Shows disk usage in blocks for /home

# change to the directory where disk.log resides
cd /var/log

# make a copy of disk.log
cp disk.log disk.log.0

# change to the target directory
cd /home

# run the du -sk * command on all files
# in /home and redirect the output
# to /var/log/disk.log
du -sk * > /var/log/disk.log

# display the output of the du -sk *
# command to standard output
cat /var/log/disk.log
```

It's not a good idea to hard-code pathnames into your scripts like we did in the previous example. We specified /var/log as the target directory several times, but what if the location of the files changed? In a short script like this one, the impact is not great. However, some scripts can be hundreds of lines long, creating a maintenance headache if files are moved. A way around this is to create a variable to take the place of the full pathname, such as:

```
LOGDIR=/var/log
```

The fourth line of the script would change from:

```
cp disk.log disk.log.0
```

to:

```
cp ${LOGDIR}/disk.log ${LOGDIR}/disk.log.0
```

Then, if the locations of disk.log changes in the future, you would only have to change the variable definition to update the script. Also note that since you are defining the pathname with the LOGDIR variable, the cd /var/log line in the script is unnecessary. Likewise, the du -sk * > /var/log/disk.log and cat /var/log/disk.log lines can substitute ${LOGDIR} for /var/log.