Introduction to UNIX: Lecture Eight

8.1 Objectives

This chapter covers:

- Shells and shell scripts.
- Shells variables and the environment.
- Simple shell scripting
- Advanced shell scripting.
- Start-up shell scripts.

8.2 Shells and Shell Scripts

A shell is a program which reads and executes commands for the user. Shells also usually provide features such job control, input and output redirection and a command language for writing *shell scripts*. A shell script is simply an ordinary text file containing a series of commands in a shell command language (just like a "batch file" under MS-DOS).

There are many different shells available on UNIX systems (e.g. `sh`, `bash`, `csh`, `ksh`, `tcsh` etc.), and they each support a different command language. Here we will discuss the command language for the Bourne shell `sh` since it is available on almost all UNIX systems (and is also supported under `bash` and `ksh`).

8.3 Shell Variables and the Environment

A shell lets you define variables (like most programming languages). A variable is a piece of data that is given a name. Once

you have assigned a value to a variable, you access its value by prepending a `$` to the name:

```
$ bob='hello world' ↵
$ echo $bob
hello world
$
```

Variables created within a shell are local to that shell, so only that shell can access them. The `set` command will show you a list of all variables currently defined in a shell. If you wish a variable to be accessible to commands outside the shell, you can *export* it into the *environment*:

```
$ export bob ↵
```

(under `csh` you used `setenv`). The environment is the set of variables that are made available to commands (including shells) when they are executed. UNIX commands and programs can read the values of environment variables, and adjust their behaviour accordingly. For example, the environment variable `PAGER` is used by the `man` command (and others) to see what command should be used to display multiple pages. If you say:

```
$ export PAGER=cat ↵
```

and then try the `man` command (say `man pwd`), the page will go flying past without stopping. If you now say:

```
$ export PAGER=more ↵
```

normal service should be resumed (since now `more` will be used to display the pages one at a time). Another environment variable that is commonly used is the `EDITOR` variable which specifies the default editor to use (so you can set this to `vi` or `emacs` or which ever other editor you prefer). To find out which environment variables are used by a particular command, consult the `man` pages for that command.

Another interesting environment variable is `PS1`, the main shell prompt string which you can use to create your own custom prompt. For example:

```
$ export PS1="(\h) \w> " ↵
(lumberjack) ~>
```

The shell often incorporates efficient mechanisms for specifying common parts of the shell prompt (e.g. in `bash` you can use `\h` for the current host, `\w` for the current working directory, `\d` for the date, `\t` for the time, `\u` for the current user and so on - see the `bash` man page).

Another important environment variable is `PATH`. `PATH` is a list of directories that the shell uses to locate executable files for commands. So if the `PATH` is set to:

```
/bin:/usr/bin:/usr/local/bin:.
```

and you typed `ls`, the shell would look for `/bin/ls`, `/usr/bin/ls` etc. Note that the `PATH` contains'.', i.e. the current working directory. This allows you to create a shell script or program and run it as a command from your current directory without having to explicitly say "*./filename*".

Note that `PATH` has nothing to with filenames that are specified as *arguments* to commands (e.g. `cat myfile.txt` would only look for `./myfile.txt`, not for `/bin/myfile.txt`, `/usr/bin/myfile.txt` etc.)

8.4 Simple Shell Scripting

Consider the following simple shell script, which has been created (using an editor) in a text file called `simple`:

```
#!/bin/sh
# this is a comment
echo "The number of arguments is $#"
echo "The arguments are $*"
echo "The first is $1"
echo "My process number is $$"
echo "Enter a number from the keyboard: "
read number
echo "The number you entered was $number"
```

The shell script begins with the line "`#!/bin/sh`" . Usually "`#`" denotes the start of a comment, but `#!` is a special combination that tells UNIX to use the Bourne shell (`sh`) to interpret this script. The `#!`

must be the first two characters of the script. The arguments passed to the script can be accessed through `$1`, `$2`, `$3` etc. `$*` stands for all the arguments, and `$#` for the number of arguments. The process number of the shell executing the script is given by `$$`. the `read` number statement assigns keyboard input to the variable `number`.

To execute this script, we first have to make the file `simple` executable:

```
$ ls -l simple↵
-rw-r--r--    1 will  finance  175  Dec 13  simple
$ chmod +x simple↵
$ ls -l simple↵
-rwxr-xr-x    1 will  finance  175  Dec 13  simple
$ ./simple hello world↵
The number of arguments is 2
The arguments are hello world
The first is hello
My process number is 2669
Enter a number from the keyboard:
5↵
The number you entered was 5
$
```

We can use input and output redirection in the normal way with scripts, so:

```
$ echo 5 | simple hello world↵
```

would produce similar output but would not pause to read a number from the keyboard.

8.5 More Advanced Shell Scripting

- `if-then-else` statements

  Shell scripts are able to perform simple conditional branches:

  ```
  if [ test ]
  then
      commands-if-test-is-true
  else
      commands-if-test-is-false
  ```

```
fi
```

The *test* condition may involve file characteristics or simple string or numerical comparisons. The `[` used here is actually the name of a command (`/bin/[`) which performs the evaluation of the test condition. Therefore there must be spaces before and after it as well as before the closing bracket. Some common test conditions are:

`-s` *file*
: true if *file* exists and is not empty

`-f` *file*
: true if *file* is an ordinary file

`-d` *file*
: true if *file* is a directory

`-r` *file*
: true if *file* is readable

`-w` *file*
: true if *file* is writable

`-x` *file*
: true if *file* is executable

`$X -eq $Y`
: true if `X` equals `Y`

`$X -ne $Y`
: true if `X` not equal to `Y`

`$X -lt $Y`
: true if `X` less than `$Y`

`$X -gt $Y`
: true `if` X greater than `$Y`

`$X -le $Y`
: true if `X` less than or equal to `Y`

`$X -ge $Y`
: true if `X` greater than or equal to `Y`

`"$A" = "$B"`
: true if string `A` equals string `B`

`"$A" != "$B"`
: true if string `A` not equal to string `B`

`$X ! -gt $Y`
: true if string `X` is not greater than `Y`

`$E -a $F`
: true if expressions `E` and `F` are both true

`$E -o $F`

true `if` either expression `E` or expression `F` is true

- `for` loops

  Sometimes we want to loop through a list of files, executing some commands on each file. We can do this by using a `for` loop:

  `for` *variable* `in` *list*
  `do`
  statements (referring to `$`*variable*)
  `done`

  The following script sorts each text files in the current directory:

```
#!/bin/sh
for f in *.txt
do
    echo sorting file $f
    cat $f | sort > $f.sorted
    echo sorted file has been output to $f.sorted
done
```

- `while` loops

  Another form of loop is the while loop:

  `while [` *test* `]`
  `do`
  statements (to be executed while *test* is true)
  `done`

  The following script waits until a non-empty file `input.txt` has been created:

```
#!/bin/sh
while [ ! -s input.txt ]
do
  echo waiting...
  sleep 5
done
echo input.txt is ready
```

You can abort a shell script at any point using the `exit` statement, so the following script is equivalent:

```
#!/bin/sh
while true
do
  if [ -s input.txt ]
    echo input.txt is ready
    exit
  fi
  echo waiting...
  sleep 5
done
```

- `case` statements

  `case` statements are a convenient way to perform multiway branches where one input pattern must be compared to several alternatives:

  `case` *variable* `in`
  pattern1`)`
  statement (executed if *variable* matches *pattern1*)
  `;;`
  pattern2`)`
  statement
  `;;`
  etc.
  `esac`

  The following script uses a `case` statement to have a guess at the type of non-directory non-executable files passed as arguments on the basis of their extensions (note how the "or" operator `|` can be used to denote multiple patterns, how "*" has been used as a catch-all, and the effect of the forward single quotes `):

```
#!/bin/sh
for f in $*
do
  if [ -f $f -a ! -x $f ]
  then
    case $f in
    core)
```

```
      echo "$f: a core dump file"
      ;;
    *.c)
     echo "$f: a C program"
      ;;
    *.cpp|*.cc|*.cxx)
      echo "$f: a C++ program"
      ;;
    *.txt)
      echo "$f: a text file"
      ;;
    *.pl)
      echo "$f: a PERL script"
      ;;
    *.html|*.htm)
      echo "$f: a web document"
      ;;
    *)
      echo "$f: appears to be "`file -b $f`
      ;;
    esac
  fi
done
```

- capturing command output

  Any UNIX command or program can be executed from a shell script just as if you would on the line command line. You can also capture the output of a command and assign it to a variable by using the forward single quotes ` `:

  ```
  #!\bin\sh
  lines=`wc -l $1`
  echo "the file $1 has $lines lines"
  ```

  This script outputs the number of lines in the file passed as the first parameter.

- arithmetic operations

  The Bourne shell doesn't have any built-in ability to evaluate simple mathematical expressions. Fortunately the UNIX `expr` command is available to do this. It is frequently used in shell scripts with forward single quotes to update the value of a variable. For example:

```
lines = `expr $lines + 1`
```

adds 1 to the variable `lines`. `expr` supports the operators `+`, `-`, `*`, `/`, `%` (remainder), `<`, `<=`, `=`, `!=`, `>=`, `>`, `|` (or) and `&` (and).

8.6 Start-up Shell Scripts

When you first login to a shell, your shell runs a systemwide start-up script (usually called `/etc/profile` under `sh`, `bash` and `ksh` and `/etc/.login` under `csh`). It then looks in your home directory and runs your personal start-up script (`.profile` under `sh`, `bash` and `ksh` and `.cshrc` under `csh` and `tcsh`). Your personal start-up script is therefore usually a good place to set up environment variables such as `PATH`, `EDITOR` etc. For example with `bash`, to add the directory `~/bin` to your `PATH`, you can include the line:

```
export PATH=$PATH:~/bin
```

in your `.profile`. If you subsequently modify your `.profile` and you wish to import the changes into your current shell, type:

```
$ source .profile
```

or

```
$ . ./profile
```

The `source` command is built into the shell. It ensures that changes to the environment made in `.profile` affect the current shell, and not the shell that would otherwise be created to execute the `.profile` script.

With `csh`, to add the directory `~/bin` to your `PATH`, you can include the line:

```
set path = ( $PATH $HOME/bin )
```

in your `.cshrc`.

(BACK TO COURSE CONTENTS)
