Lab 4

Shell Scripts

Introduction to Shell Scripts

- On the most basic level, a shell script is simply an ordinary text file that contains a set of shell commands. You can use vi to create a basic shell script:

  ```
  $ vi cmd_test          (use vi to add the following lines)
  pwd
  date
  echo hello world
  ```

- Although we will learn several ways to execute shell scripts, the most basic way to execute it is with the ksh command as below:

  ```
  $ ksh cmd_test
  /u/ds59478/stuff
  Sun Jan  2 04:00:55 EST 2000
  hello world
  ```

- Essentially what we have created is a custom new UNIX command that executes three other commands. Although this example is not a particularly useful command, it provides the basis for shell scripts.

Executable Shell Scripts

- On the previous page, we created a shell script called cmd_test that is a custom UNIX command. Like other UNIX commands, we should be able to execute it without the ksh command:

```
$ cmd_test
ksh: cmd_test: 0403-006 Execute permission denied.
$
```

- However, by default, ordinary files are not created with the execute permission bit turned on:

```
$ ls -l cmd_test
-rw-r--r--   1 ds59478  dasd          26 Jan  2 04:00 cmd_test
$
```

- To make the cmd_test file executable, use the chmod command:

```
$ chmod +x cmd_test
$ ls -l cmd_test
-rwxr-xr-x   1 ds59478  dasd        26 Jan  2 04:00 cmd_test
$ cmd_test
/u/ds59478/stuff
Sun Jan  2 05:40:43 EST 2000
hello world
$
```

- Shell scripts that are to be used again should have the permissions set to executable, and should be moved to the ~/bin directory.

Shell Script Variables

- To demonstrate the use of variables in shell scripts, we shall modify the contents of our test script:

```
$ vi cmd_test
```
(use vi to change file to lines below)

```
pwd
date
x='hello world'
echo $x
```

- In this script, we assign the value of hello world to a variable called x. In the syntax of shell scripting, the equal sign (=) represents assignment and the string is represented by the single quotes (‘’). The echo command uses the ($) to represent the value of a variable as its string. Below is the result:

```
$ cmd_test
/u/ds59478/stuff
Sun Jan  2 06:00:48 EST 2000
hello world
$
```

- In this script, the $x was replaced with the hello world string.

Shell Processes

- Now, notice what happens when we look at the $x variable:

```
$ echo $x

$
```

- The script on the previous slide set the value of $x to hello world, but it is now empty. To see why, add the following statement (by using vi) to the cmd_test script:

```
echo $$
```

- The $$ parameter of echo displays the process id number:

```
$ cmd_test
/u/ds59478/stuff
Sun Jan  2 06:13:57 EST 2000
hello world
294170
$ echo $$
337662
$
```

- Notice that the process id numbers are different. This is because the command executes its own process called a **child process**. This is different from the original process which is called the **parent process**.

Sourcing Shell Processes

- In general, UNIX commands create their own child processes for the life of the command. We see this when we execute the cmd_test script via ksh or directly. However, there is a way to execute the command in our current shell process. This is done by using the source (.) command as follows:

```
$ . cmd_test
/u/ds59478/stuff
Sun Jan  2 06:25:25 EST 2000
hello world
337662
$ echo $x
hello world
$ echo $$
337662
$
```

- As you can see the process ids are the same for the cmd_test as they are for the shell session.   Also, the shell now has the $x variable set to hello world. This is known as “sourcing” a command.

More Shell Processes – ps command

- In general, at any given time, UNIX systems have many processes running on them at once (multi-user, multi-threaded). To view all of these processes, we can use the **ps** command with the –ef flag:

```
$ ps –ef
```

- To view your processes in this format, you can pipe the output of the ps command to a grep for your id:

```
$ ps -ef | grep ds59478
 ds59478 102222 337662   0 06:41:56  pts/2  0:00 grep ds59478
 ds59478 140262 337662  24 06:41:56  pts/2  0:00 ps -ef
 ds59478 337662 379778   0 05:33:50  pts/2  0:00 -ksh
$
```

- At the time this command was running, I had only my native (or parent) shell process going. The other two entries are for the ps and grep commands themselves.

- There are many commands that revolve around the management of processes that are outside the scope of this course. However, learning the ps command and using echo $$ provide an introductory look at processes.

Shell Environment Variables – set command

- In previous labs, we have seen a special file called ~/.profile which contains user defined defaults for the shell environment. There are actually several different files that serve to customize the user environment. We covered the ~/.profile because it is a common convention used in all UNIX.

- Specifically on AIX systems, there are several files such as /etc/environment, /etc/passwd, /etc/groups and /etc/profile that help define the default user environment. Generally, users can not edit these files without the help of a system administrator. However, these files contain the default definitions of several environment variables.

- The **set** command displays all of the variables that are defined to the shell. When you first log into a system, this includes all of the environment variables that get initialized. If you define a new variable to the shell, then it also can be displayed by using the set command. Below is an example:

```
$ x=matt

$ set | grep x

x=matt

$
```

Shell Environment Variables – PATH

- One of the most basic and common uses of shell programming is creating and/or modifying the shell environment variables via the ~/.profile file. For example, all variables set in the /etc/profile file can be overridden by changing them in the ~/.profile of the user.

- An example of this is the PATH variable. This variable determines the directory paths that the shell will search when attempting to execute a command. The following command allows you to view your PATH variable:

```
$ echo $PATH
```

- Binary executable programs are normally stored in a bin directory. When users write a script that will be used again, they will often move the script to the ~/bin directory. If it is a program that could be used by other users, then it can be moved to a shared bin directory such as /usr/bin.

- In the example below, we create a new script in the ~/bin directory and source it. The system knows where to find the script because of the PATH variable:

```
$ echo ‘clear;ls –al’ > ~/bin/clr

$ . clr
```

Shell Environment Variables – PS1 and export

- Another shell variable is PS1, which controls the shell prompt. Some users (especially those coming from a DOS background) would like to see the current directory as part of the shell prompt. This is done with the PS1 variable as below:

  ```
  PS1='$PWD  $ '
  ```

- As we learned before, shell variables only exist for the current (or parent) shell process. If subsequent (or child) processes are started, then the shell variables need to be re-assigned. However, this can be avoided with the **export** command:

  ```
  export PATH PS1
  ```

- The export command passes the shell variables to the child process so that the shell variables do not need to be reassigned. This process is aptly called **inheritance**, because parent shells pass on their values to the child shells.

- At this point, we can see that the ~/.profile is not just a special file, but a working shell script that assigns variables, passes parameters and executes UNIX commands.

Interpreted Programming Languages – sed, awk and perl

- As mentioned earlier, the UNIX programming environment supports interpreted programming language scripts. Three very popular languages in the UNIX environment are sed, awk and perl. Below is a brief explanation of each:
  - sed – stands for stream editor. It is mostly used for repetitive changes in text patterns as a "find and replace". It can be issued interactively from the vi editor or via the command line.
  - awk – named for its authors Aho, Weinberg and Kernighan of Bell Labs. Specializes in formatting text from multiple input sources. Serves as a great tool for report generation. Very similar to C language in structure (developed by the same people).
  - perl – stands for practical extraction and reporting language, developed by Larry Wall. Full blown programming language that combines the features and functions of C, sed, awk and shell programming. There are many different ways to perform the same function using perl. Available on multiple platforms as public domain software.

- There are entire books and courses dedicated to each one these languages. However, as an introduction, in this class we will write a very simple script in each language.

An example of sed – text substitution

- People most commonly use sed for global substitutions in text files. Below is a simple example:

```
$ cat syllabus
This file contains the spring syllabus for MGT6346
which will be taught in the spring semester.
This spring, I will spring into action as
I teach this class in the spring.
$ sed s/spring/summer/ syllabus > summer_syllabus
$ cat summer_syllabus
This file contains the summer syllabus for MGT6346
which will be taught in the summer semester.
This summer, I will spring into action as
I teach this class in the summer.
$
```

- Notice that only one occurrence of “spring” was substituted on each line. To substitute all occurrences, specify the letter **g** in the criteria for a **global** substitution:

```
$ sed s/spring/summer/g syllabus > summer_syllabus
```

An example of awk – formatting the output of ls

- In the UNIX environment, awk (using the –f option) is often used to take the output from a command and format it:

```
$ cat awk_ls
BEGIN {print "Bytes" "\t" "Filename"} (sets up header)
{sum += $5;print $5 "\t" $9} (Loops through input)
END {print "Total Bytes are "sum} (sets up footer)
$ ls -l | awk -f awk_ls
Bytes   Filename
101     awk_ls
172     summer_syllabus
172     syllabus
15      test
Total Bytes are 460
$
```

- Although awk is still commonly used, awk programs that interact with UNIX commands are generally being replaced by the more robust perl language. The availability of awk to perl conversion utilities has helped to facilitate this migration.

An example of perl –interactively listing home directory files

- An example of how perl can interact with commands (as we have seen with awk) and with UNIX commands (as we have seen with shell scripting) can be seen in the program below:

```
$ cat perl_dir
#!/usr/bin/perl
print "Enter the username of the home directory you would like to view: ";
chop($hdir = <STDIN>);
chdir(~$hdir) || die "Invalid username"
foreach(<*>) {
print "$_\n";
}
$
```

- A topic within the world of perl that is outside the scope of this course (but worth mentioning) is the use of macros. One of the things that makes perl so powerful is that perl macros can be written on one platform and ported to others. The use of perl macros is one of the reasons that it is becoming a popular scripting language for web-based applications.