Lab 5

The UNIX Programming Environment

Introduction to Programming in UNIX

- One of the great strengths of the UNIX operating system is its ability to support programming. The UNIX environment supports the following types of programming:
  - Shell Programming
    - Programming shell scripts with the syntax and commands that are supported by a native UNIX shell. Examples include sh, ksh, csh, bash and many others.
  - Interpreted Programming Languages
    - Programming in a language syntax that is fed into a larger UNIX program (known as an interpreter) and executed by that program. Examples include sed, awk and perl.
  - Compiled Programming Languages
    - Programming in a language syntax that is compiled into object code and run as a binary executable program. Examples include C, C++, FORTRAN, PASCAL, assembler, etc.
- In this section we will expand on the shell programming concepts introduced in Lab 4. We will also introduce the interpreted languages of sed, awk and perl.

Shell Programming – using arguments

- In the last section, we introduced the concept of shell scripts. We can also use the shell as a programming environment by taking advantage of the constructs that are built into the shell scripting language. In this section, we will learn some of these basic constructs to expand on our shell programming abilities.

- Also, in the last section, we learned about shell variables. These are variables that are defined within the shell script. **Arguments** are special shell variables that get passed to the program at the execution of the command. In many of the UNIX commands we have learned, we have been passing arguments to programs. If you recall, the standard format of a command is:

  `$ command –options argument1 argument2` ... etc

- In order to use arguments, the format is simply ${n} with n being the position of the argument in the command. Consider the following simple program below:

```
$ cat arg_script
echo I have a dog named $1
echo I have a spouse named $2
echo I have a dad named $3
$ ksh arg_script daisy india jack
I have a dog named daisy
I have a spouse named india
I have a dad named jack
$
```

Shell Programming – read statement

- Arguments are very useful for programming in the context of UNIX-style commands. However, like any programming language, shell scripts can also read input interactively from users. In the example below, the same program that read arguments now presents an interactive dialogue with the user:

```
$ cat read_script
echo 'What is the name of your dog?'
read dog
echo 'You better take' $dog 'for a walk!'
$ ksh read_script
What is the name of your dog?
daisy
You better take daisy for a walk!
$
```

- The read command utilizes standard input. By default, as in this example, this means input from the screen. However, this can be redirected to use input from other sources such as a file.

- By using the expr statement, simple integer arithmetic can be performed inside a shell program. For an example, we can write a relatively simple and useful shell script that will convert blocks of disk to megabytes. In AIX, a block of disk is 512 bytes. Two blocks are equal to 1024 bytes, which is also known as a kilobyte. Therefore, if we divide the number of blocks by 2048, then we get the number of megabytes. The script below does this:

```
$ cat conv
expr $1 / 2048
$ conv 10000
4
$
```

- A few things of note about the script above. First, I used the chmod +x command to make the file conv executable so that I did not need to use the ksh command to execute it. Also, the expr returns an integer number, so the resulting number in megabytes is rounded.

Shell Programming – Conditional Operations using if

- The conv command from the last exercise is somewhat useful in aiding the calculation of disk space on AIX systems. However, let's say that this program could work better if it could convert either from blocks to megabytes or vice-versa.

- In order to do this, we will need to calculate the formula depending upon the type of input. This requires what is known as conditional processing. The most simple form of conditional processing is the **if** statement.

- The format of the if statement is as follows:

  `if test` condition

  `then`

  statements (executed only on true condition)

  `else`

  statements (executed only on false condition)

  `fi`

- Note: If statements can be nested to check for multiple conditions. This will be done in the updated conv program on the next page.

Shell Programming – Updated conv program using if statements

- The updated conv program uses nested if statements to check for blocks, mb, or invalid data:

```
$ cat conv
if test "$1" = "blocks"
 then
  expr $2 / 2048
 else
 if test "$1" = "mb"
  then
   expr $2 \* 2048
  else
   echo You have entered invalid data:
   echo 'The format is: conv [blocks|mb] number'
 fi
fi
$ conv blocks 10000
4
$ conv mb 4
8192
$ conv foo
You have entered invalid data:
The format is: conv [blocks|mb] number
$
```

Shell Programming – A look at the ~/.profie script

- We can also use the if statement in the ~/.profile script. In the statement below, the script checks for the existence of new mail and executes a mail session if there is some:

```
$ cat ~/.profile

PATH=/bin:/usr/local/bin:/usr/bin:/sbin:_

/usr/local/etc/httpd:/etc:/usr/ucb:$HOME/bin:/usr/bin/X
11:.

export PATH

TERM=ibm3151

if [ `tty` = /dev/tty0 ]

then

     export TERM=ibm3151

else

     export TERM=vt100

fi

set -o vi


if [ -s "$MAIL" ]

then echo "$MAILMSG"

fi


EDITOR=/usr/bin/vi

export EDITOR

PS1='$ '
```

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.