LECTURE - 18

References : Sumitabha Das

SECTION -D

SHELL PROGRAMMING

References : Sumitabha Das

INTRODUCTION

- Shell programming
- Pattern matching and Wild cards
- Functions
- Shell types
- Shell Variables

INTRODUCTION

- When you log on to a UNIX machine, you first see a prompt. This prompt remains there until you key in something. Even though it may appear that the system is idling, a UNIX command is in fact running at the terminal. But this command is special ; it’s with you all the time and never terminates until you logout. This command is **shell**.

SHELL OFFERINGS

Your UNIX system offers a variety of shells for you to choose from. Over time, shells have become more powerful by the progressive addition of new features. The shells we consider in this text can be grouped into two categories :

- The Bourne family comprising the Bourne shell (/bin/sh) and its derivatives – the Korn shell (/bin/ksh) and Bash (/bin/bash).
- The C Shell (/bin/csh).

The absolute pathname of the shell's command file is shown in parenthesis. Everything that applies to Bourne also applies to its supersets, Korn and Bash.

PATTERN MATCHING - THE WILD-CARDS

- We begin with the special set of characters that the shell uses to match filenames. We've already used commands with more than one filename as arguments (e.g. **cat chap01 chap02**).
- Often we may need to enter multiple filenames in a command line. To illustrate this point, try listing all filenames beginning with chap. The most obvious solution is to specify all the filenames separately :

```
$ ls chap    chap01  chap02    chap03   chap04   chapx chapy chapz
```

- If filenames are similar (as above), we can use the facility offered by the shell of representing them by a single pattern or model. For instance , the pattern chap* represents all filenames beginning with chap. This pattern is framed with ordinary characters (like chap) and a meta character (like *) using well defined rules.
- The pattern can then be used as an argument to the command, and the shell will expand it suitably *before the command is executed*.

- The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called **wild-cards**(something like a joker that can match any card).

The Shell's Wild-Cards

Wild – Card	Match
*	Any number of characters including none.
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of the characters x and z.
[!ijk]	A single character that is not an i, j, or k (Not in C shell).
[!x – z]	A single character that is not within the ASCII range of the characters x and z (Not in C shell.)

(1) THE * AND ?

- Previously we've used the command **ls chap*** to list some filenames beginning with **chap.** The metacharacter, * , is one of the characters of the shell's wild-card set.
- It matches any number of characters (including none). It thus matches all filenames specified in the previous command line which can now be shortened in this way:

```
$ ls chap *
 chap    chap01  chap02    chap03   chap04   chapx chapy chapz
```

- Observe that chap* also matching the string chap. When the shell encounters this command line, it identifies the * immediately as a wild-card. It then looks in the current directory and recreates the command line as below from the filenames that match the pattern chap* :

```
$ ls chap    chap01  chap02    chap03   chap04   chapx chapy chapz
```

- The shell now hands over t his command to the kernel which uses its process creation facilities to run the command.

- The next wild-card is ?, which matches a single character. When used with same string chap (chap?), the shell matches all five-characters filenames beginning with chap. Appending another ? Creates the pattern chap??, which matches six-character filenames. Use both expressions separately, and the meaning becomes obvious :

```
$ ls chap?
 chapx   chapy  chapz

$ ls chap??
 chap01   chap02  chap03
```

(2) THE CHARACTER CLASS:

- The pattern framed in previous examples are not very restrictive. With the knowledge we have, its not easy to list only chapy and chapz. Nor is easy to match only first four chapters from the numbered list. You can frame more restrictive patterns with the **character class.**
- The character class comprises a set of characters enclosed by the rectangular brackets , [and], but it matches a single character in the class. The pattern [abcd] is a character and it matches the single character – an a , b, c or d. This can be combined with any string or another wild-card expression, so selecting chap01, chap02, chap03 now becomes a simple matter :

```
$ ls    chap0[123]
 chap01   chap02   chap03
```

- Range specification is also possible inside the class with a – (hyphen) ; the two characters on either side of it form the range of characters to be matched. Here are two examples:

$ ls chap0[1-3] *Lists chap01, chap02, chap03*

$ ls chap[x-z] *Lists chapx, chapy, chapz*

- **<u>Negating the Character Class (!):</u>** How about framing a pattern that reverses the above matching criteria? The solution that we prescribe here unfortunately doesn't work with the C- shell, but with the other shells, you can either use the ! as the first character in the class to negate the class. The two examples below should make the point clear:

***. [!c o]** Matches all filenames with a single-character extension but not the .c or .o files.

[! a – z A-Z] Matches all filenames that don't begin with an alphabetic character

General shell functions :

- The UNIX shell program interprets user commands, which are either directly entered by the user, or which can be read from a file called the shell script or shell program.
- Shell scripts are interpreted, not compiled.
- The shell reads commands from the script line per line and searches for those commands on the system, while a compiler converts a program into machine readable form, an executable file - which may then be used in a shell script.
- Apart from passing commands to the kernel, the main task of a shell is providing a user environment, which can be configured individually using shell resource configuration files.

SHELL TYPES

Just like people know different languages, your UNIX system will usually offer a variety of shell types:

1. **sh or Bourne Shell:** the original shell still used on UNIX systems and in UNIX-related environments. This is the basic shell, a small program with few features. While this is not the standard shell, it is still available on every Linux system for compatibility with UNIX programs.
2. **bash or Bourne Again shell:** the standard GNU shell, sensitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user. On Linux, **bash** is the standard shell for common users. This shell is a so-called *superset* of the Bourne shell. This means that the Bourne Again shell is compatible with the Bourne shell: commands that work in **sh**, also work in **bash**. However, the reverse is not always the case.

4. **csh or C shell:** the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.
5. **tcsh or TENEX C shell:** a superset of the common C shell, enhancing user-friendliness and speed. That is why some also call it the Turbo C shell.
6. **ksh or the Korn shell:** sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration a nightmare for beginning users.

(**Briefs about GNU O.S.** :- GNU is a Unix-like operating system created and funded by the Free Software Foundation. One of the goals of the Free Software Foundation was an operating system composed entirely of free software. Many pieces, such as shells, utilities, and compilers were created for this purpose. The GNU operating system has yet to be fully completed, due to slow development and debates about design goals. Most of the programs created for GNU have been ported to other kernels and operating systems, most notably Linux.)

- The file /etc/shells gives an overview of known shells on a Linux system:

```
xyz:~>cat /etc/shells
 /bin/bash
/bin/sh
/bin/tcsh
/bin/csh
```

- To switch from one shell to another, just enter the name of the new shell in the active terminal. The system finds the directory where the name occurs using the PATH settings, and since a shell is an executable file (program), the current shell activates it and it gets executed. A new prompt is usually shown, because each shell has its typical appearance:

```
xyz:~>  tcsh
[xyz@post21 ~]$
```

The Bourne shell programming:

Unix runs Bourne shell scripts when it boots. If you want to modify the boot-time behavior of a system, you need to learn to write and modify Bourne shell scripts.

What's it all about?

First of all, what's a shell? Under Unix, a shell is a command interpreter. That is, it reads commands from the keyboard and executes them. Furthermore, you can put commands in a file and execute them all at once. This is known as a script. Here's a simple one:

```
#!/bin/sh # Rotate procmail log files
cd /homes/arensb/Mail
 rm   procmail.log.6                          # This is unnecessary
mv   procmail.log.5    procmail.log.6
  mv    procmail.log.4     procmail.log.5
 mv   procmail.log.3     procmail.log.4
 mv    procmail.log.2     procmail.log.3
mv   procmail.log.1      procmail.log.2
mv    procmail.log.0     procmail.log.1
mv    procmail.log       procmail.log.0
```

There are several things to note here: first of all, comments begin with a hash (#) and continue to the end of the line

- Secondly, the script itself is just a series of commands. I use this script to rotate log files, as it says. I could just as easily have typed these commands in by hand, but I'm lazy, and I don't feel like it. Plus, if I did, I might make a typo at the wrong moment and really make a mess.

```
#!/bin/sh
```

- The first line of any script must begin with #!, followed by the name of the interpreter.
- A script, like any file that can be run as a command, needs to be executable: save this script as rotatelog and run

```
$ chmod +x rotatelog
```

to make it executable. You can now run it by running

```
$  ./rotatelog
```

Unlike some other operating systems, Unix allows any program to be used as a script interpreter. This is why people talk about ``a Bourne shell script'' or ``an awk script.'' One might even write a *more* script, or an *ls* script (though the latter wouldn't be terribly useful). Hence, it is important to let Unix know which program will be interpreting the script.

- When Unix tries to execute the script, it sees the first two characters (#!) and knows that it is a script. It then reads the rest of the line to find out which program is to execute the script. For a Bourne shell script, this will be /bin/sh. Hence, the first line of our script must be

#!/bin/sh

- After the command interpreter, you can have one, and sometimes more, options.
- Once Unix has found out which program will be acting as the interpreter for the script, it runs that program, and passes it the name of the script on the command line.
- Thus, when you run **./rotatelog**, it behaves exactly as if you had run **/bin/sh ./rotatelog.**

Shell Variables:

- sh allows you to have variables, just like any programming languages. Variables do not need to be declared. To set a sh variable, use

```
VAR=value
```

and to use the value of the variable later, use

`$VAR` or

`${VAR}`

The latter syntax is useful if the variable name is immediately followed by other text:

```
#!/bin/sh COLOR=yellow
 echo This looks $COLORish
 echo This seems ${COLOR}ish
```

prints

```
This looks
This seems yellowish
```

There is only one type of variable in sh: strings. This is somewhat limited, but is sufficient for most purposes.

LOCAL VS. ENVIRONMENT VARIABLES

- A sh variable can be either a local variable or an environment variable. They both work the same way; the only difference lies in what happens when the script runs another program (which, as we saw earlier, it does all the time).
- Environment variables are passed to subprocesses. Local variables are not.
- By default, variables are local. To turn a local variable into an environment variable, use

export VAR

HERE'S A SIMPLE WRAPPER FOR A PROGRAM:

```
#!/bin/sh
NETSCAPE_HOME=/usr/imports/libdata
CLASSPATH=$NETSCAPE_HOME/classes
export CLASSPATH
$NETSCAPE_HOME/bin/netscape.bin
```

Here, NETSCAPE_HOME is a local variable; CLASSPATH is an environment variable. CLASSPATH will be passed to *netscape.bin* (*netscape.bin* uses the value of this variable to find Java class files); NETSCAPE_HOME is a convenience variable that is only used by the wrapper script; *netscape.bin* doesn't need to know about it, so it is kept local.

The only way to unexport a variable is to unset it:

```
unset VAR
```

APPLICATIONS

- Many shell script interpreters double as command line interface, such as the various Unix shells, Windows PowerShell or the MS-DOS COMMAND.COM. Others, such as AppleScript or the graphical Windows Script Host (WScript.exe), add scripting capability to computing environments without requiring a command line interface. Other examples of programming languages primarily intended for shell scripting include DCL and JCL.

RESEARCH

- **Shell-Storm.org** is a development organization based on **GNU/Linux** systems that provide **free projects** and **source codes**. **Shell-storm.org** provides useful information to people who perform **security** testing.
- **Title:** Easy way to bypass randomization by chaining ret2ret and ENVAR (without NX) **Language:** English **Author:** Florian Gaultier (agix) (twitter) **Date** 2011-04-28
- Website : http://research.shell-storm.org/files/research-12-en.php