SOFTENGER
Strive to Excel

Shell

The Shell’s Interpretive Cycle

- Whenever you log on to a Unix machine you first see a prompt .
- This prompt remains there until you key in something .
- Even though the system appears to be idle , a UNIX command is still running at the terminal .
- This command remains with you until you logout .
- This command is the shell .

- The Shell’s Interpretive Cycle
  - The moment you key in something the shell swings in too action.
  - Run the command :

```
$ps
PID     TTY      TIME    CMD
1486    pts/2    0:00    bash        bash shell running
```

The Shell’s Interpretive Cycle

When You key in the command it goes as input to the shell.

•The shell first scans for the metacharacters. (ex > , | ,*)

•It perform all actions represented by these characters before the command can be executed.

•Ex.   rm *

1. * makes no sense to the rm .
2. the shell replaces it with all filenames in the current directory
3. rm ultimately runs with these filenames as arguments .

The Shell’s Interpretive Cycle

- Ex .cat > foo
- The > means nothing to cat , so the shell creates the file foo and connects cat’s output to it. When all pre-processing is complete , the shell passes on the command line to the kernel for execution .
- The command line has none of the metacharacters that were originally seen by the shell.
- The shell waits for the command to complete and normally can’t do any work while the command is running.
- After the command execution is complete , the prompt reappears and the shell returns to its waiting role to start the next cycle.

- Different types of shell
- Unix offers variety of shells .
- Roughly they can be categorized into two .
- The Bourne shell (/bin/sh) and its derivatives – the Korn shell (/bin/ksh) and Bash (/bin/bash).
- The C shell (/bin/csh) and its derivatives , Tcsh (/bin/tcsh).
- echo $SHELL will display the absolute pathname of the shell’s command file .

Wild Cards	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character either i , j or k .
[x-z]	A single character within the ASCII range of x and z.
[!ijk]	A single character that is not an i , j , k . (Not in C Shell.)
[!x-z]	A single character not within the ASCII range of x and z.
{pat1,pat2}	pat1,pat2 (Not in Bourne Shell.)

$ ls chap * displays all file in the directory starting with chap.. including chap.

$ echo * displays list of all files in the current directory.

$ ls .???* lists all hidden files in your directory having at least three characters after the dot .

Escaping

- When we use the \ immediately before a metacharacter , it turns of its special meaning .
- The \ tells the shell that the metacharacter has to be matched literally instead of interpreting it as metacharacter.
- The \ suppresses the wild-card nature of the * , thus preventing filename expansion of it .
- Ex $ rm chap*   will remove the file chap* and not the files chap1 ,chap2 ,etc .
- \ is used to escape special characters like space characters also .
- Ex $ my\ document.doc
- Ex $ echo \\ will output \ . The backslash is used to escape itself.

Quoting

- Another way to turn off the metacharacter is quoting.

Ex . echo ‘\’ displays a \
rm ‘chap*’ removes file chap*.

- Single Quotes protects all special characters (except the single quote).
- Double Quotes are more permissive ; they don’t protect the $ and the ` (backquote)

Ex echo “total files -`ls -l | wc -l`” will output
total files – 30
and echo ‘total files -`ls -l | wc -l`’ will output
total files -`ls -l | wc -l`

The Three Standard Files

- The Shell associates three files with the terminal , two for display and one for keyboard.
- Shell makes this file available as soon as the user logs in.

- Standard Input – The file (or stream) representing the input , which is connected to keyboard.
- Standard Output - The file (or stream) representing the output , which is connected to display.
- Standard Error – The file (or stream) representing error messages that emanate from the command shell . This is also connected to display.

The Three Standard Files

- Commands don’t usually write to terminal . They perform all terminal-related activity with these three files that shell makes available to every command.
- These special files are actually streams of character which many command sees as input and output.
- A stream is a sequence of bytes .
- Every command that uses streams will always find these files open and available .
- These files are closed when user logs out .
- Even though the shell associates each of these files with a default physical device , this association is not permanent .
- The shell unhooks a stream from its default device and connect it to a disk file (or to any command) the moment it sees some special characters in the command line .

The Standard Input

- Cat and wc commands are used to read disk files.
- These commands have an additional method of taking input.
- When these commands are used without arguments they read the file representing the standard input.
- Standard Input has three input sources
- The Keyboard , the default source.
- A File using redirection with the < symbol (a metacharacter).
- Another program using a pipeline.

Ex.

```
$ wc
$ wc < sample.txt
$ ls | wc
```

- $ wc < sample.txt
- Here wc doesn’t open sample.txt.
- It reads the standard input file as a stream but only after the shell made a reassignment of this stream to a disk file.
- wc has no idea where the stream came from and it is also not aware that the shell has opened the file sample.txt on it’s behalf !
- $ wc sample.txt
- In this case the sample.txt is opened by the program wc and not the shell .

The Standard Input

- You can also take input from both file and standard input.
- cat - foo first from standard input then from foo
- cat foo - first from foo and then from standard input

- All commands displaying output on the terminal actually write to the standard output file as a stream of characters .
- There a three possible destination of the streams
- The Terminal , the default destination
- A file using the redirection symbol > and >>
- As input to another program using a pipeline

The Standard Output

- $ wc sample.txt > newFile

- The sequence of execution works like this :
- On seeing the > , the shell opens the disk file newFile for writing.
- It also unplugs the standard output file from the default source and reassigns it to newFile .
- Next , wc (and not the shell) opens the file sample.txt for reading and writes to standard output which has earlier been reassigned by shell to newFile .
- Any command that uses standard output is ignorant about the destination of its output also .

The Standard Output

- If the output file doesn’t exist the shell creates it before executing the command .
- If it exists the shell overwrites it .
- Using >> symbol you can append to the file .
- $ wc sample.txt >> newFile

The Standard Error

- When you enter an incorrect command or try to open a nonexistent file , certain diagnostic messages show up on the screen . This is standard error whose default destination is terminal .
- Trying to “cat” a nonexistent file produces the error stream.
- $ cat foo

cat : cannot open foo

- cat fails to open the file , and writes to standard error.

The Standard Error

- $ cat foo > errorfile
- The diagnostic output has not been snbt to errorfile .
- Standard error cannot be redirected in the same way the standard output can (with > or >>).
- Even though standard error and standard output use the terminal as default destination , the shell possess a mechanism to capture them individually . Redirecting standard error requires the use of 2 > symbol.
- $ cat foo 2>errorfile
- $ foo.sh   > bar1 2> bar2.

Filters Using Both Standard input and Standard Output

- Unix command can also be grouped into four categories .
- Commands that take neither standard input nor standard output . Ex cp , mv , rm , mkdir , rmdir , cd .
- Commands that don’t take standard input but they send their output to standard output . Ex ls , pwd , who .
- Commands that takes standard input but no standard output Ex lp.
- Commands which takes both standard input and standard output Ex cat , wc , od , cmp .
- Commands in fourth categories are called filters

- Filters Using Both Standard input and Standard Output

```
$ cat calc.txt
        2  ^ 10
        25 * 25
        30 * 25 + 15 ^ 2

$ bc < calc.txt > result.txt
        $ cat result.txt
        1024
        625
        975
```

- bc obtained the expression from redirected standard input , processed them and sent out the result to a redirected output stream .

Shell variables

- Shell variables are integral part of shell program.
- They provide ability to store & manipulate information within a shell program.

RULES OF NAMING A VARIABLE

1) Can be a combination of alphabets ,digits & an underscore
2) No commas & blanks are allowed.
3) The first character must be an alphabet or an underscore.
4) The variable name should be of any reasonable length

Direct Assignment

One can declare & initialize a variable in one shot as

$ name=SEaD or “SEaD”

$ age=25

Important Tips

1) While assigning the values you should not leave blank space on either side of = sign.

2) The shell variables are string variables i.e in the statement age=25 the number 25 is stored as string.

3) A variable can contain more than 1 word but in that case the whole string should be enclosed in quotes

4) We can carry out more than one definitions on a line

```
$ a=25     b=Anil       c=“My name is”
```

4) We can enclose the variables in literals

```
$ echo $c   $b  &  my age is  $a
```

5) All the variables declared inside a shell die the moment the execution of the shell is over.

6) A variable which defined but not given any value is known as NULL variable.

7) If a NULL variable is enclosed any where in command shell manages to ignore it.for e.g. $ var1=“ “ var2=“ “

```
$ wc -c $var1 $var2  myfile
800
$
```

Here the var1 & var2 are not considered to count the characters.

8) Using set command we can display not only system variables but also user defined variables.

9) We can make a variable unchangeable as

```
$ a=90
$ readonly a
```

10) We can wipe out the declared variables by unsetting them.

for e.g. $ unset a

This will unset (wipe out) the variable ‘a’ & we can use this name to declare the variable. Obviously the environment variables can not be unset by a normal user.

export command

- Set the value of a variable so it is visible to all subprocesses that belong to the current shell.
- Ex

```
$ PAGER=less
$ export PAGER
$ echo $PAGER
```

Escape Mechanism

Escape Mechanism	Effect
\ (Backslash)	Negates special properties of single character following it: * is literal asterisk.
‘ ‘ (pair of single quotes)	Negates special properties of all enclosed characters: ‘Take this *$?# sentence literally.’
“ “ (Pair of double quotes)	Negates special properties of all enclosed characters except $, ` , \ : “The value of the rent is $rent.”

- The single quotes cause the back quotes and the & to be printed literally.
- The double quotes also cause the & to be printed literally, but the $rent is replaced by its output.
- Once you use an opening single or double quote, the shell expects you to provide a closing quote, too.
- If you hit Return before doing so, the shell shifts to its second prompt, telling you it expects more to the command.This gives you a means to print several lines with a single echo.

Dot command

- Executing Commands from a file : .(dot)
- A filename started with .(dot) (containing UNIX commands) reads and executes commands .
- Standard shell scripts cause a new subshell to be created to run the script The dot command uses the same shell.
- It just uses the redirection to take the commands from the file.
- A script executed via the dot command can change the value of shell variable in the current shell.

Co-processes - ksh only

- Adding the operator |& (pipe ampersand) after a command or program will run it as a co-process in the background.
- It may be easier for you to remember this syntax if you think about what these characters represent individually. | creates a pipe, and & starts jobs in the background.
- Using the -p option with the print and read commands will write to and from a co-process

Example :

```
#!/bin/ksh
while [ 1 ]
do
  read   arg
  echo   $arg   $$
done
```

```
    $ ./script_1 |&
```

run the script

$$ displays the pid of the current process.

- The next sequence of commands will write to and read from the co-process, and then display the returned value to stdout:

```
•                    $ print -p hello
•                    $ read -p line
•                    $ echo $line
•                        hello 124
```

- The string hello was passed to the co-process with the print statement, and then read -p was used to capture the returned value in the variable line. Examine the value of line shows that the co-process converted "hello" to hello followed by the pid of the shell.

History command

Example :

history	show list of commands in history
!-2	current line -2
! 21	21st command
!string	refer to most recent command starting with string
?string[?]	refer to most recent command containing string.
!102:s/dir1/dir2	substitute dir1 with dir2 in 102th command
!1029:s/tmp/root/	substitute tmp with root
!1055:s/dir2/”//root”	substitute dir2 with /root
!1055:s/java/\/root/	substitute java with /root