Shells

Shell definition

Interface between the user and the operating system

Different Shells

Bourne (`sh`), `Korn (ksh), C Shell (csh),`

`Bourne Again Shell (bash), tcsh, zsh`

Kinds of shells

- Bourne Shell (sh)

This is the standard shell that is part of most UNIX operating systems. It's normally the default shell on the system

- Korn Shell (ksh)

This is a superset of the Bourne shell. It has the basic features of the Bourne shell as well as others. One can run normally most scripts written for sh under ksh

- C Shell (csh)

Developed by Uni. Of California, part of Berkley Software Distribution version of UNIX. The syntax different from sh and ksh as it is a C language - styled syntax

Kinds of shells continued

•Bourne Again Shell (bash)

Linux comes with its own standard shell, this is based on the Bourne shell.

Changing Shells

type at the prompt the shell to execute and exit to return to the default shell

Assuming ksh is the default shell to change to bash and then back

$bash↵

bash$ exit ↵

Running commands

By default commands run in the foreground. The shell waits for the termination of the command before returning to the prompt, giving the user the chance to type another command.

To run a command in the background (so that the shell will return immediately without having to wait for the command to finish), the ampersand symbol (&) has to be used in the end of the command line.

Example:
% emacs test.txt &

The emacs editor will start and the shell will wait straight away to the prompt, without waiting for emacs to exit.

Interrupting and killing processes

When a process is run in the background, it cannot be terminated using CTRL-C.

The kill command can be used to send signals to processes (including the termination signal), by specifying their process id (pid).

The -9 signal can be specified with kill. This is the “strongest” request to “kill” a process, as this signal cannot be blocked.

The ps (process status) command lists the processes currently running.

The echo command

The echo utility displays the text that the shell passes to it, followed by newline.

Example:

```
% echo 'hello there'
hello there
%
```

Equivalently one can have:

```
% echo "hello there"
hello there
% echo hello there
hello there
```

This is equivalent to us using

%printf "%s\n" "string"

%printf "%s\n" "hello there"

Built in Shell Commands

The shell command interpreters (sh, ksh, bash etc..) have built in functions, which are interpreted by the shell as commands

These commands are part of the shell itself

Command	**Shells:**		
	Bourne Shell	Korn Shell	Bourne Again Shell
alias	Not available	alias ls1="ls –l"↵	
echo	echo Hi, "\n" this is a test ↵		
history	Not available	history [no. or letter or string]↵	
kill	kill –9 jobID ↵		
set	set ↵ displays set values | var=value ↵ | echo $var ↵		
unalias	Not available	unalias label ↵ e.g unalias ls1 ↵	
unset	unset var_label ↵		

To specify multiple commands in the command line

The semicolon ; can be used to separate commands in one line:

% ls ; cat file.txt ; ls /lib

The above executes the three processes sequentially. The output of each command will be displayed one after another. I.e ls 1st then cat and lastly ls /lib command

To execute the above processes concurrently:

```
% ls & cat file.txt & ls /lib
```

output can come out in any order and possibly mixing the output, i.e. some of the output of the first process might be printed followed, by the output of the second, followed by some more output of the first, etc.

Grouping commands

Parentheses can be used to group commands.

The shell creates a new shell (a subshell), for each group, treating each group of commands as a seperate job.

The shell forks processes to execute the commands.

Example

Execute commands a and b sequentially in the background, while executing c in the foreground:

```
% (a ; b) & c
```

Creating a Script

To create a script showing all the users currently logged and the current date.

Use a text editor to type the following and save the file

```
date
echo Users currently logged in
who
```

Then we need to run the script

To do this we need to set the files properties so that it is executable

We use the chmod unix command; we type

chmod +x filename ↵

To execute type ./filename s↵

In UNIX, if a file is marked as executable, and begins with a line that looks something like

#!/bin/ksh #!/bin/sh #!/bin/bash

This specifies which shell the commands will be executed in. Once finished the system will return back to the standard shell

Note the # character appearing at the beginning of the line will make the shell ignore that line

So we can use this to place comments into our scripts

The Shell Built in Commands

Command	Built into		
	Bourne shell	**Korn Shell**	**Bourne Again Shell**
exit	Available	Available	Available
for	Available	Available	Available
if	Available	Available	Available
let		Available	Available
read	Available	Available	Available
test	Available	Available	Available
until	Available	Available	Available
while	Available	Available	Available

The following script will display the current date and then the number of users logged on to the system followed by the pathway for the current working directory

```
#!/bin/bash
#
# displays the current date and time
# and the number of users logged on
# and the full path of the current working directory
#
date  #display the current date
who | wc –l  #display no of users logged in use wc -l counts no. of lines
pwd  #display current working directory
```

The output of the script would something like

```
11 tiger% ./path.sh
Wednesday February  1 14:54:35 GMT 2006
      32
/home/camel/u2/staff00/charalg
12 tiger%
```

An improved version

```
#!/bin/bash
#
# displays the current date and time
# and the number of users logged on
# and the full path of the current working directory
#
echo #skip a line
echo –e "Date and time:\c" #print message using \c to inhibit the next line feed
date
echo –e "Number of users on the system: \c"
who | wc -l # displays the no. of users logged on
echo –e "Your current working Directory: \c "
pwd        # displays the current directory
echo    #throw a line
```

Which generates the following output

```
27 tiger% ./path2.sh

Date and time:Wednesday February  1 15:14:14 GMT 2006
Number of users on the system:       18
Your current directory: /home/camel/u2/staff00/charalg

28 tiger%
```

The echo Command Special characters

Escape Character	Definition
\b	Backspace
\c	Inhibit the line feed at the end of output string
\n	A carriage return and a line feed
\r	A carriage return only
\t	A tab character
\0n	A zero followed by 1- 2- or 3-digit octal number representing the ASCII code of a character

The Read Command

Syntax read [option] Var Var…

Option

Use –p “prompt message” as option to prompt user

Consider the following script

```
#!/bin/bash
#
# using the read command
# prompt user for name and then print it out
#
echo                                # skip a line
echo –e ″Enter your name: \c″  # prompt the user
read  name                          # read from standard input and save into var name
echo –e ″Your name is $name″        #echo back the inputted line
echo                                # skip a line
```

```
#!/bin/bash
#
# using the read command with prompt
# prompt user for name and then print it out
#
echo                                    # skip a line
read -p "Enter Your Name " name   # prompt and  read  and save into var name
echo "Your name is $name"          #echo back the inputted line
echo                                    # skip a line
exit
```

Which generates the following output

```
28 tiger% ./read.sh

Enter your name: George↵
Your name is George

29 tiger%
```

Note its good practice to put variables in quotation marks, because one can not predict the users input and we don't want the shell to interpret special character such as *, ? And so on

Example 2 using read with more than one argument

```
#!/bin/bash
#
# using the read command
# prompt user for name and address and then print it out
#
echo                                    # skip a line
echo –e ″Enter your name and address: \c″    # prompt the user
read  f_name s_name address          # read from standard input and save into vars
echo ″Your name is $f_name $s_name ″ # echo back the 1st  two strings
echo ″Your Address is $address ″            # echo back the contents of var address
echo                                 # skip a line
```

Which generates the following output

```
29 tiger% ./read2.sh


Enter your name and address:  George Charalambous 21 London  Rd Harrow↵
Your name is George Charalambous
Your Address is 21 London  Rd Harrow


30 tiger%
```

The read f_name s_name address command will read in three strings
To manage the whole input read will read the 1st two words into the
1st two variables and the rest into the last variable.
The 1st two vars. The read command will use the blank space as the delimiter.
But for the last var it will use the CR
So var address will hold 21 London Rd Harrow

Variables

To store the value into a variable use the assignment operator =

variable= value

Note spaces are not permitted either side of the equal sign

UNIX shell does not support data types it will interpret any value as a string of characters

e.g count=1 means store the character 1 into var count

variables in shell script will stay in memory until the script is finished. To remove them while executing a script use

unset var_lablel

Variables names

Variable names must begin with a letter or the underscore _

You can use numbers or letters for the rest of the variable name

Do not use * ? \ & $ @ “ characters as they may cause problems

Passing the output of a Command into a variabe

To do this you must enclose the command in a pair of grave accent marks `

The following script will call upon the command date and then pass the output string into a variable called date1 and then the contents of the variable will be displayed on screen

```
#!/bin/bash
date1=`date` # copy the output of date to date1
echo "the date is $date1” # output date1
```

Command Line parameters

Shell scripts can read upto 10 command line parameters from the command line into the special vars.

Note the command line arguments are the items typed after the command, separated by spaces

The special vars are numbered 0 to 9 and are named $0, $1 .. $9

The Shell positional variables

Variable	Definition
$0	Contains the name of the script
$1,$2…$9	Contains the 1st through to the 9th command line parameters
$#	Contains the number of command line parameters
$@	Contains all command line parameters “$1 $2…$9”
$?	Contains the exit status of the last command
$*	Contains all command line parameters “$1 $2…$9”
$$	Contains the PID number of the executing process

Using the shell special variables

```
#!/bin/bash
# A sample script to show the shell vars
echo
echo "The script name is $0"
echo "The number of arguments is $#"
echo "The 1st parameter is $1"
echo "The 2nd parameter is $2"
echo "The list of parameters is $@"
echo "The list of parameters is $*"
echo "The PID no of current process is $$"
echo "the exit status of last command is $?"
echo
echo "bye"
```

If we execute the script with no arguments we have

```
[root@localhost /root]# ./test3.sh



The script name is ./test3.sh
The number of arguments is 0
The 1st parameter is
The 2nd parameter is
The list of parameters is
The list of parameters is
The PID no of current process is 502
the exit status of last command is 0
```

bye

```
[root@localhost /root]#
```

If we execute the script with arguments we have

```
[root@localhost /root]# ./test3.sh hello this is a test


The script name is ./test3.sh
The number of arguments is 5
The 1st parameter is hello
The 2nd parameter is this
The list of parameters is hello this is a test
The list of parameters is hello this is a test
The PID no of current process is 504
the exit status of last command is 0


bye
[root@localhost /root]#
```

Using the set command

One can use the set command to assign values to the positional variables

The following script will assign three values to three positional variables and then display them on screen

```
#!/bin/bash
date
set house car `date`
echo "$1"
echo "$2"
echo "$3"
echo "$4"
echo "$5"
echo "$6"
echo
```

Generates the following output

```
[root@localhost /root]# ./test4.sh
Thu Feb  2 16:51:26 UTC 2006
house
car
Thu
Feb
2
16:51:26

[root@localhost /root]#
```

The following script copy the file and will then open the file with emacs editor and then exit

```
#!/bin/bash
cp $1 prevF       #copy the file to the named prevF
nano $1          #invoke emacs with the filename held by$1
exit 0         #exit the program with the signal of 0
```

Condition statements

if then statement

```
if [ condition ]
then
    commands
fi
```

The if statement ends with the reserved fi word (if backwards)

The square brackets around the condition are necessary and must be surrounded by white spaces or tabs

The following script copy the file if filename provided and will then open the file with emacs editor and then exit

```
#!/bin/bash
if [ $# = 1 ]     #check the no. of command line args
then
 cp -f $1 prevF  #copy the file to the named prevF
fi          #end of if
nano $1           #invoke emacs with the filename held by$1
exit 0         #exit the program with the signal of 0
```

Test if no of argument is 1 i.e. a filename is provided if so make the back up copy use -f option for the copy command that will overwrite the file without warning and then open the file with the editor.

Test command numeric operators

-eq equal to

-ne not equal to

-lt less than

-le less than or equal to

-gt greater than

-ge greater than or equal to

if then else construct

```
if [ condition ]
then
    true-command(s)
else
    false-commands(s)
fi
```

Don’t forget the need for the space between the [and] characters

The following script will check the hour of the time and if before 12pm will display the message good morning else good afternoon

```
#!/bin/bash
midday=12
hour=`date +%H`
echo "$hour" "$midday"
if [ "$hour" -lt "$midday" ]
then
   echo "Good Morning"
else
   echo "Good Afternoon"
fi
echo
exit
```

```
hour=`date +%H`
```

We execute the date command with the option +%H where we limit the output to just the hour part of the date

so

$date ↵

Wed Nov 30 14:00:52 EDT 2005

$date +%H↵

14 …… show only the hour of the day

$

Other fields +%M … minutes

+%S …. seconds

The if then elif Construct

This is the nested if the syntax for which is:-

```
if [ condition_1 ]
then
        commands_1
elif [ condition_2 ]
then
        commands_2
elif [ condition_3 ]
then
        commands_3
else
        commands_n
fi
```

```
#!/bin/bash
midday=12
evening=18
echo
hour=`date +%H`
echo "$hour" "$midday"
if [ "$hour" -lt "$midday" ]
then
   echo "Good Morning"
elif [ "$hour" -lt "$evening" ]
then
   echo "Good Afternoon"
else
   echo "Good Evening"
fi
echo

exit 0
```

The test Command

It evaluates an expression passed to it as an argument and returns 0 if true and a non zero value if false

```
#!/bin/bash
# script using test
#
echo
echo "Are you OK?"
echo -e "Input Y for yes and N for no: \c" #prompt user
read answer
if test "$answer" = Y # equi if [ "$answer" = Y ]
then
   echo "Glad to hear that!"
else
  echo "I'm sorry to hear that!"
fi
echo
exit 0
```

The test command string test operations

=	equal to	“$str1” = “$Str2”
!=	not equal to	“$str1” != “$Str2”
-n	test if string contains char	-n “$string”
-z	test if string empty	-z “$String”

Rem test will return 0 if true and non zero val if false

Logical operators

test expression_1 logical_operator expression_2

-a and: test returns 0 if true if both expression are true

-o or: test returns 0 if one or both expression are true

! not: test returns 0 if expression is false

Example 2 of the use of test with logical and

```
#!/bin/bash
# script using test
# reads three numbers and then prints out largest
echo
echo "Please type in three numbers: \c"
read val1 val2 val3
if test "$val1" -gt "$val2" -a "$val1" -gt "$val3"
then
  echo "The largest number is: $val1"
elif test  "$val2" -gt "$val1" -a "$val2" -gt "$val3"
then
   echo "The largest number is: $val2"
else
  echo "The largest number is: $val3"
fi
echo
exit 0
```

Test command for files

-r	-r filename	does filename exist is it readable
-w	-w filename	does filename exist is it writable
-x	-x filename	does filename exist is it executable
-s	-s filename	does filename exist, and does it have non zero length
-f	-f filename	does filename exist, and not a directory
-d	-d filename	does filename exist, is it a directory

The following script tests if a given file is read and writable

and displays if so

```
#!/bin/bash
# script using test
#
echo
echo
if test $# = 1 #test that parameter typed
then
  if test -w "$1" -a -r "$1"
  then
     echo "$1 has both read and write permissions"
  fi
else
  echo "you must specify a filename"
fi
echo
exit 0
```

Arithmetic Operations

Need to use expr command

expr int1 + int2	integer addition
expr int1 - int2	integer subtraction
expr int1 / int2	integer division
expr int1 * int2	integer multiplication
expr int1 \% int2	returns remainder

Example of use of expr

```
#!/bin/bash
x=10   #assign x to 10
y=10   #assign y to 10
x=`expr $x + 1` #increment value of x by 1
echo ”$x”
if test $x -lt "$y"  #test if var x is less than y
then
 echo "$x less than $y"
else
 echo "$y less than $x"
fi
exit 0
```

let

Use let to carry out arithmetic operations

The let command automatically uses the values of the variables. No need to type $var_label use variable_label

The let command interprets the * and % without the need for * or \% respectively

Example using let command

```
#!/bin/bash
read -p "Type 1st number " x
read -p "Type 2nd Number" y
let x=x+1 # use let to increment x
echo "$x"
if test $x -lt "$y"
then
 echo "$x less than $y"
else
 echo "$y less than $x"
fi
exit 0
```

The for-in statement

Syntax:

```
for loop-index in argument-list
do
   commands
done
```

This iterates and executes the commands as many times as the number of items in the argument-list.

In each iteration, the loop-index is assigned the value of one of the arguments in argument-list in successive order.

Example
% cat fruit.bash

```
#!/bin/bash
for fruit in apples oranges pears bananas
do
echo $fruit
done
```

```
% ./fruit
apples
oranges
pears
bananas
```

The for statement

Syntax:

```
for loop-index
do
   commands
done
```

The above structure iterates and executes the commands as many times as the number of command line arguments.

The loop-index takes on the value of each of the command line arguments, one at a time.

Example:

```
$ cat for_example2.sh
#!/bin/bash
for i
do
  echo $i
done
```

```
$ ./for_example2.sh a b c d e
a
b
c
d
e
```

The while statement

```
while [ condition ]
do
commands
done
```

Iterate and execute commands until the condition is false, in which case the loop terminates.

Example

Write a shell script which displays all the integers which are less than 5.

```
#!/bin/bash
number=0
while [ "$number" -lt 5 ]
do
echo "$number "
number=`expr $number + 1`
done
```

Note ` is the grave accent not the single quote

```
$ ./count
0
1
2
3
4
```

The until statement

Syntax:

```
until [ condition ]
do
commands
done
```

The commands are executed iteratively and while the condition is false.
As soon as the condition evaluates to true, the loop exits.

Example

Write a script which prompts the user to guess a secret name, known only by the script itself.

```
#!/bin/bash
secretname=Bob
name=noname
echo "Try to guess the secret name!"
echo
until [ "$name" = "$secretname" ]
do
echo -n "Enter your guess: "
read name
done
echo "Good guess!"
```

Example – Improved version limiting the no. of guesses

```
#!/bin/bash
secretname=Bob
name=noname
try=0
Mattempt=3
echo "Try to guess the secret name!"
echo
until [ "$name" = "$secretname" -o $try -eq
$Mattempt ]
do
  try=`expr $try + 1`
  echo -n "attempt $try Enter your guess: "
  read name
done
if [ $name = $secretname ]
then
 echo  "Good Guess“
else
 echo "Good ran out of attempts"
fi
```