CISC 151
The UNIX Operating System

Lecture 7

Richard L. Holladay, CCNA, Ph.D.

Lecture 7

Shell Programming

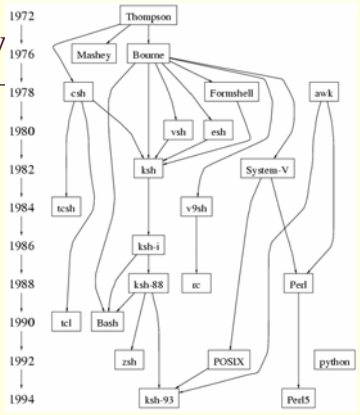
What is a shell?

- The user interface to the operating system
- Functionality:
  - Execute other programs
  - Manage files
  - Manage processes
- Full programming language
- A program like any other
  - This is why there are so many shells

3

Shell History

- First shell (sh) written by Ken Thompson in 1972
- Shell features evolved as UNIX grew



Most Commonly Used Shells

/bin/csh	C shell
/bin/tcsh	Enhanced C Shell
/bin/sh	The Bourne Shell / POSIX shell
/bin/ksh	Korn shell
/bin/bash	Korn shell clone, from GNU

- All UNIX systems include C shell and its predecessor, the Bourne shell
- bash probably the most popular

5

Other Shells

- **MH shell (msh)**: add email access.
- **Job Control Shell (jsh):** a version of Bourn shell includes C shell-like job control.
- **Remote Shell or Restricted Shell (rsh)**: for sites that have security considerations.
- **Secure Shell (ssh)**: is intend to replace **rsh**. It has strong authentication and secure connections over unsecured channels.
- **Portable Operating System Interface (POSIX)**: Perl interface to IEEE Std 1003.1, a superset of the Bourne shell.
- **Z Shell (zsh)**: include almost all good feature of other shells.

6

The Shell of Linux

- Linux has a variety of different shells:
  - Bourne shell (sh), C shell (csh), Korn shell (ksh), TC shell (tcsh), Bourne Again shell (bash).
- Certainly the most popular shell is bash.
- Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh).
- It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard.
- It offers functional improvements over sh for both programming and interactive use.

7

Choose Your Working Shell

- First check out the available shells in the system
- Check the current working shell
- Switch between shells:
  - **csh**: to switch to C shell
  - **ksh**: to switch to K shell
  - **bash**: to switch to B again shell
  - **tcsh**: to switch to terminal based C shell
  - **sh**: to switch to the Shell

8

Identifying Your Shell

- **ls -lFd /bin/*sh*** lists various shells in the system.
- **ls -lFd /usr/local/bin/*sh*** list the various shells of the local system.
- **echo $SHELL** or **grep 'yourid' /etc/passwd**: to find out you default shell.

9

Identifying Your Shell

- **$$**: to display the process identifier **pid** of your current shell.
- **ps**: to display your current running process.
- **ps –p $$**: to display your current shell.
- **ps -ef | grep $$**: to identify your current working shell.
- Check the symbols to identify your working shell. **%** means csh or tcsh. **$** means bsh or ksh. But they might be different.
- **chsh**: to change your shell. (Not available in our CSE)

10

Ways to use the shell

- **Interactively**
  - When you log in, you interactively use the shell
- **Scripting**
  - A set of shell commands that constitute an executable *program*

11

Functions and Features of Shells

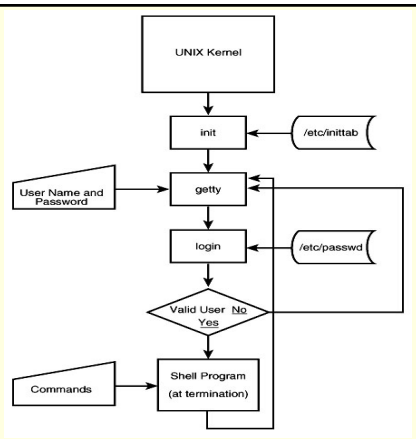
- Command-line interpretation
- Reserved words
- Shell meta-characters (wild cards)
- Access to and handling of program commands
- File handling: input/output redirection and pipes
- Maintenance of variables
- Environment control
- Shell programming

12

How the Kernel and the Shell Interact

- All shells run programs the same way
  - A sort executed from csh works the same way a sort executed from bash does
- **Kernel** of UNIX has been loaded from disk to memory when UNIX has been brought on line.
- **Kernel** will remain there until you turn off the machine.
- Program **init** runs as a background and remains running until shutdown. The process flow from the kernel through the login process is like the following figure:

13



The Shell and the Child Process

- After you finish logging on, the shell program layer is in direct contact with the kernel.
- If you enter command such as “ls”, the shell locates the actual program file “/bin/ls” and passes it to the kernel to execute.
- The **Kernel** then create a new child process area, load the program and executes the instruction.
- After the process complete, the **kernel** recovers the process area and returns control to the parent shell program

15

Auto-Execution of the Shell

- Some UNIX resources can execute a shell program without human interaction, which we can Auto-Execution.
- Example cron (clock daemon)
- To use this feature, the user need to specify the shell to run in the first line of the shell program.
- Example: $! /bin/sh

16

The Shell Environment

- **env | cat -n**: to list the various aspects of your working environment.
- Variable Description
  - **HOME**: This is your home directory obtained from the fourth field of the password file
  - **SHELL**: kind of shell you are using
  - **TERM**: Your terminal definition. You can define it to the appropriate value within .login file at your home directory
  - **USER**: Your user ID information.

17

The Shell Environment

  - **PATH**: The directories definition so that a command can be searched through them.
  - **MAIL**: Your email storage place
  - **LOGNAME**: The synonym of the user
  - **NAME**: Your real name information storage place.
- You can personalize your shell environment by define anything in the environment. Certain programs can read many environment variables that customize their behaviour.

18

Shell Configuration Files

- **.profile**: for **bash** shell
- **.bashrc**: for **bash** shell.
- **.cshrc**: for **csh** shell.
- **.login**: for **tcsh** shell (default shell of cse).
- **.plan**: for user information look up (finger) command.

19

Shell Programming

- As well as using the shell to run commands you can use its built-in programming language to write your own commands or programs.
- Creating and executing the shell script:
  - Use a text editor to create a file:
    ```
    vi filename
    ```
  - Define execute permission:
    ```
    chmod u=rwx filename
    ```
  - Execute the script
    ```
    filename
    ```

20

Origin of Scripting Languages

- Scripting languages originated as *job control languages*
  - 1960s: IBM System 360 had the Job Control Language
  - *Scripts* used to control other programs
    - Launch compilation, execution
    - Check return codes
- Scripting languages got increasingly more powerful in the UNIX world
  - Shell programming, AWK, Tcl/Tk, Perl
  - *Scripts* used to combine *components*

21

System Programming Languages

- System programming languages replaced assembly language
  - Benefits:
    - The compiler hides unnecessary details, so these languages have a higher level of abstraction, increasing productivity
    - They are *strongly typed*, *i.e.* meaning of information is specified before its use, enabling substantial error checking at compile time
    - They make programs more portable

22

Higher-level Programming

- Scripting languages provide an even higher-level of abstraction
  - The main goal is programming productivity
    - Performance is a secondary consideration
  - Modern SL provide primitive operations with greater functionality
- Scripting languages are usually interpreted
  - Interpretation increases speed of development
    - Immediate feedback
  - Compilation to an intermediate format is common

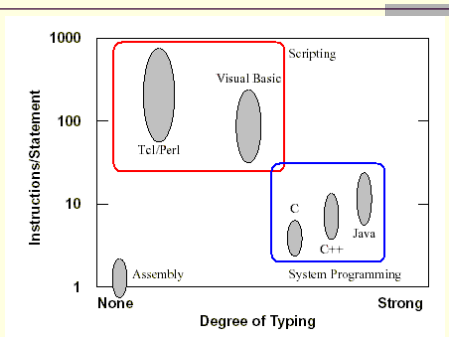
23

Higher-level Programming

- They are *weakly typed*
  - *i.e.,* Meaning of information is inferred
  - × Less error checking at compile-time
    - Run-time error checking is less efficient, but possible
  - ✓ Weak typing increases speed of development
    - More flexible interfacing
    - Fewer lines of code
- They are not usually appropriate for
  - Efficient/low-level programming
  - Large programs

24

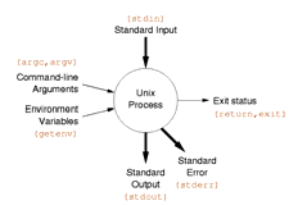
Typing and Productivity



25

UNIX Programs

- **Means of input**:
  - Program arguments [control information]
  - Environment variables [state information]
  - Standard input [data]
- **Means of output**:
  - Return status code [control information]
  - Standard out [data]
  - Standard error [error messages]



26

Shell Scripts

- A shell script is a regular text file that contains shell or UNIX commands
  - Before running it, it must have execute permission:
    - **`chmod +x filename`**
- A script can be invoked as:
  - **`ksh name [ arg … ]`**
  - **`ksh < name [ args … ]`**
  - **`name [ arg …]`**

27

Shell Scripts

- When a script is run, the **kernel** determines which shell it is written for by examining the first line of the script
  - If 1st line starts with **`#!pathname-of-shell`**, then it invokes *pathname* and sends the script as an argument to be interpreted
  - If **`#!`** is not specified, the current shell assumes it is a script in its own language
    - leads to problems

28

Simple Example

```
#!/bin/sh

echo Hello World
```

29

Scripting vs. C Programming

- Advantages of shell scripts
  - Easy to work with other programs
  - Easy to work with files
  - Easy to work with strings
  - Great for prototyping. No compilation
- Disadvantages of shell scripts
  - Slow
  - Not well suited for algorithms & data structures

30

Programming or Scripting ?

- Difference between programming and scripting languages:
  - Programming languages are generally a lot more powerful and a lot faster than scripting languages.
  - Programming languages generally start from source code and are compiled into an executable. This executable is not easily ported into different operating systems.

31

Programming or Scripting ?

- A scripting language also starts from source code, but is not compiled into an executable.
  - Rather, an interpreter reads the instructions in the source file and executes each instruction.
  - Interpreted programs are generally slower than compiled programs.
  - The main advantage is that you can easily port the source file to any operating system.
- bash is a scripting language.
  - Other examples of scripting languages are Perl, Lisp, and Tcl.

32

Overview of Shell Scripts

- At their simplest, shell scripts are are just an ASCII file with Unix commands in them.
- *Comments* can be in a shell script.
  - A comment is a '#' anywhere on a line and everything following to the end of the line.
- Shell scripts are fed, one line at a time, to a particular shell and *interpreted* by that shell as it sees the commands.

33

Using UNIX Shell Scripts

- Shell scripts can be a very effective tool for prototyping applications, or in many cases, developing production ready applications
- You can write shell scripts that present user-friendly screens and perform most of the functions that you would expect from an application written in a compiled language such as “C”.
- Shell scripts also save you time by automating long command sequences that you must perform often
  - good for administration tasks, even if you’re not the Administrator

34

Activity 1 – A First Shell Script

- Create a new file named hello with vi
  ```
  $ vi hello
  ```
- Specify the shell
  ```
  #!/bin/sh
  ```
- Write the rest of the script
  ```
  #!/bin/sh
  echo Hello!
  # Use the echo and whoami commands
  # with command substitution to
  # print the username
  echo I am `whoami’
  ```
- Save file and Exit vi (<esc> :s :q)
- Make your script executable
  ```
  chmod u=rwx hello
  ```

35

Activity 1 – A First Shell Script

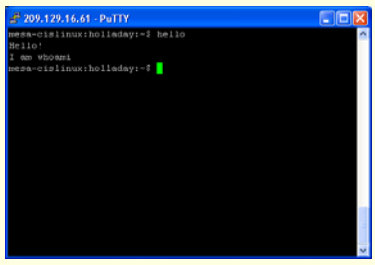
- Run script:
  ```
  $ hello
  ```

“I am whoami” not exactly what we wanted!

Why?

Used ‘ under “ on the keyboard (single quotes)

Needed ` under ~ (back ticks)
More later…

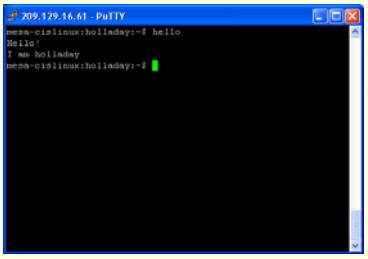


36

Activity 1 – A First Shell Script

- Fix and Run again:

```
$ hello
```



37

Shell Script Development Cycle

1. Decide what the script will do.
2. Make a list of commands.
3. Create a new file for the script.
4. Identify the shell the script will use.
5. Add commands and comments.
6. Save the script file.

Creation of the Shell Script.

7. Make the script file executable.
8. Type the name of the script to execute it.

Executing the Shell Script

9. Debug and modify the script if errors occur.

Debugging the Shell Script

38

Creating the Shell Script

- '.sh' is the conventional filename extension.
- Use `#!/path/to/shell` on the first line of the script to execute the script with the desired shell. Otherwise, the parent shell is used.
  - `#!/usr/sh`
  - `#!/usr/ksh`
- Avoid the names of Unix commands for your filenames.

39

Saving Scripts

- Create your own library scripts to save you typing and time
- Typing a full pathname for these would be tedious
  - Create a personal bin directory to store their shell scripts:

```
$ mkdir ~/bin
PATH=${PATH}:~/bin
```

40

Executing a Shell Script

- A shell script is always run in a sub shell (new process spawned).
- You can run a shell script in two ways:
  - *`shell script_name`* at the command line.
  - Make the script executable, then just use the name of the script like a Unix command.

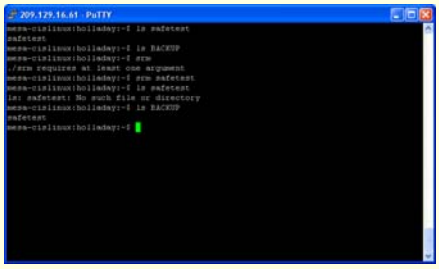
41

Example: srm - Safely Remove a File

```
#!/bin/sh
# Script srm - Safely remove a file
# Takes at least one argument (name of file)
# Backs up file. Will only delete file if backup OK
if [ $# -eq 0 ]; then
  echo $0 requires at least one argument
  exit
fi
while ( [ $# -gt 0 ] )
do
  cp $1 $HOME/BACKUP
# $? returns exit status of last command
  if [ $? -eq 0 ]; then
    rm $1
  fi
# shift gets the next argument, if any
shift;
done
```

42

Test of srm Script



43

Components of Shell Scripts

- Shell Variables
- Operators
- Logic Structures

44

Variables

- Variables are symbolic names that represent values stored in memory
- The three types of variables discussed in this section are:
  - configuration variables
  - environment variables
  - shell variables

45

Configuration Variables

- Use configuration variables to store information about the setup of the operating system, and do not change them
- Configuration variables bear standard names, such as HOSTNAME
- Configuration variables are CAPITALIZED to distinguish them from user variables

46

Environment Variables

- Set up environment variables with initial values that you can change as needed
- UNIX reads these variables when you log in
- They determine many characteristics of your session
- Environment variables bear standard names, such as HOME, PATH, SHELL, USERNAME, and PWD
- Environment variables also CAPITALIZED to distinguish them from user variables

47

Environment Variables

- To use environment variables:

```
cd $HOME
myPath=$HOME
HOME="/home/richard"
echo $HOME
```

48

.profile

- The individual users UNIX version of "autoexec.bat"
  - Sets environment variables
  - Defines shell prompt
  - Launches applications
  - Sets path
- To see the .profile type:
  `more .profile`
  while in the $HOME Directory

49

Default .profile

- A generic `.profile` is sometimes found in the `/etc/skel` directory
- When new accounts are created, copy the `.profile` from `/etc/skel` to the users $HOME directory and update the contents

50

Customizing Your Prompt

- Your login prompt is configured automatically at login in **`.profile`**
- To see how it is set: type "**`echo $PS1`**" and press enter
  - [\d \t \u \h \s \w \W]
  - \d = date \t = time \u=user \h=host name
  - \w = path of working directory
  - \W = name of the working directory w/o path
- To set your prompt
  `PS1="\d \t \W >"`
- Default prompt on buffy:
  `PS1="\h:\u:\W$"`

51

Alias

- Allows you to create a shorthand for frequently used commands
- Put them in the .profile file

```
alias newname="unixcommand"
```

- Stays in effect until session ended or it is unaliased

52

Shell Variables

- Shell variables are those you create at the command line or in a shell script
- Variables are not typecast
  - Shell programming languages do not use typed variables, so the same variable can be used to store integers one time and a string the next

53

More Shell Variables

- User-Created Shell Variables

  string variables

  myname=Richard

  numberinclass=12
- Notice there are no spaces between the =operator and its operands
- Echo will show you the value of a variable

```
echo $myname
```

- To use the variable in an expression

```
YourVar=$MyVar
```

54

Storing Values

- If you want a variable to contain the ***result*** of a UNIX command, use the ` (back tick) to enclose the command:

```
mywords=`wc –w mytext`
```

- To set a variable to a string of characters containing spaces:

```
MEMO="Meeting will be at noon today"
echo $MEMO
```

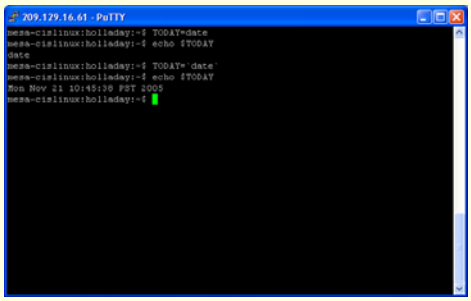
55

Activity: Using the Back Tick (`)

- Type `TODAY=`date`` and press Enter.
- This command:
  - creates the variable `TODAY`
  - executes the date command
  - stores the output of date in the variable `TODAY`
- No output appears on the screen
- Type `echo $TODAY` and press Enter.
- You see the output of the date command that was executed above.

56

Activity: Using the Back Tick (`)



57

Using Quotes: ' vs. "

- ' stops the shell from interpreting the value of a variable

```
test="Wrong Answer"
echo "This is a $test"
```

- Will print:

```
This is a Wrong Answer
```

- However:

```
echo 'This is a $test'
```

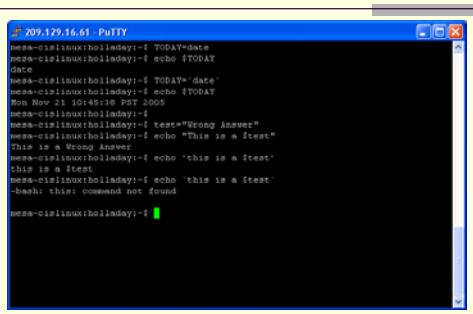
does not return the value of test

- Will print:

```
This is a $test
```

58

Using Quotes: ' vs. "



59

'Quotes', "Quotes", or `Back Ticks`?

Quote Character	Meaning	Example Command / Result
Single quote (') - must occur in pairs	Tells the shell to display anything in single quotes, including metacharacters, literally. Does not allow for variable expansion.	$echo "*** $LOGNAME ***" *** $LOGNAME ***
Double quote (") - must occur in pairs	Tells the shell to display any metacharacters literally. Allows for variable expansion.	$echo "*** $LOGNAME ***" *** user10 ***
Back quotes (`) - must occur in pairs	Tells the shell to display the output of the command instead of interpreting it literally.	$echo `uname -n` sunblade1

60

More Variables

- To list your environment variables:
  - The list of environment variables probably spans more than one screen, so use the more command with printenv or the set command.

```
printenv | more
```

Or: `set | more` and press Enter

  - Press the spacebar to page through the output

61

Exporting Shell Variables to the Environment

- Shell scripts cannot automatically access variables created on the command line or by other shell scripts
  - Shell variables are local variables
  - Only known to the shell that created it
- To make a variable created on the command line, or in another shell script generally available, you must use the "export" command to make it an environment variable

```
export $MyVar
```

- `export` with no arguments will tell you what has already been exported

62

Variables – Samples

```
$ CAT=kate          First CAT gets its value
$ echo $CAT
kate
$ sh                Create new shell
$ echo $CAT
                    Echo response to undefined variable
$CAT=munchkin       A second CAT gets its value
$ echo $CAT
munchkin
$ ctrl-D
$ echo $CAT         Return to old shell
kate                Doesn't know about new shell's CAT
$
```

63

Variables – More Samples

```
export CAT
$ CAT=kate          First CAT gets its value
$ echo $CAT
kate
$ sh                Create new shell
$ echo $CAT
kate                The new shell has a copy of the 1st CAT
$ CAT=munchkin      A second CAT gets its value
$ echo $CAT
munchkin
$ ctrlD$ echo $CAT  Return to old shell
kate                Doesn't know about new shell's CAT
$
```

64

Special Variables

- Null string
  - `emptystring=''`
  - `emptystring=`
  - `emptystring=""`

65

Interactive Scripts

- To prompt the user to enter a value use the "read" command: `read username`

```
echo Dear User, what is your name
read name
echo Glad to meet you, $name
```

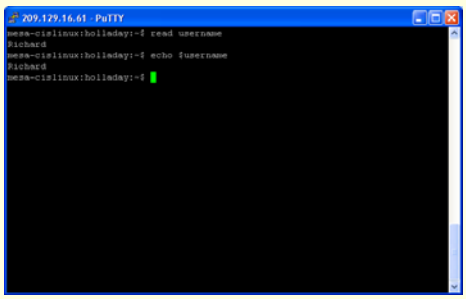
66

Activity: Interactive Scripts

- At the prompt type `read username`
- press return – and enter a name – press enter again to get back to the prompt
- Type `echo $username`

67

Activity: Interactive Scripts



68

Conditional Operators

- || (OR)
- && (AND)

```
cc $* && a.out
```

- Compiles the list of files given as command line arguments.
- If successful then executes the compiled version

69

Using the Test Command

- The test command
  - makes preliminary checks of the UNIX internal environment and other useful comparisons (beyond those that the if command alone can perform)
  - can place the test command inside your shell script
  - or execute it directly from the command line
  - uses operators expressed as options to perform the evaluations

70

Using the Test Command (cont'd)

- The test command can be used to:
  - Perform relational tests with integers (such as equal, greater than, less than, etc.)
  - Test strings
  - Determine if a file exists and what type of file it is
  - Perform Boolean tests

71

Shell Operators

- The Bash shell operators are divided into three groups:
  1) Defining and evaluating operators
  2) Arithmetic operators
  3) Redirecting and piping operators

72

Operators

- String Checking – String Tests
- Numerical – Relational Integer Tests
- Logical – Testing Files

73

Operators - String

- String Checking

```
string1 = string2
string1 != string2

test $ans = yes (spaces are required)
test –n string   True if the string has nonzero length
test –z string   True if the string has zero length
test string      True if the string is not a null string
```

74

Operators – String Sample

```
$ frog=bumpy
$ test $frog ; echo $?
0
$ test $frag ; echo $?
1
$
```

0 usually means FALSE
WHY does this return 0 ????

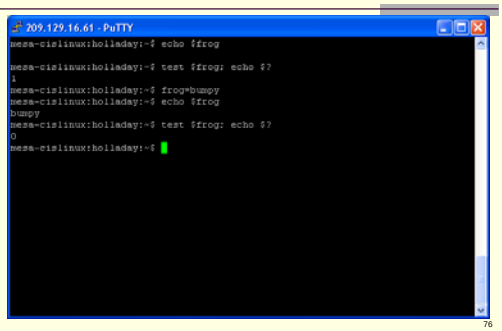
This was a typo – frag doesn't exist so WHY does this return 1 (TRUE) ????

So what's going on here??

- Test returns the value of the test in the variable ?
- You can see the result of the test by looking at ? with *echo* (must use $? since ? is a variable)
- Here test is using the –n option by default (test if the length of a string is nonzero) – See: man test

75

Operators – String Sample



76

Arithmetic Operators

- Arithmetic operators consist of
  - + for addition
  - - for subtraction
  - * for multiplication
  - / for division
- When using arithmetic operators, the usual mathematical precedence rules apply:
  - multiplication and division are performed before addition and subtraction

77

Example Arithmetic Operators

- Create a simple shell script
- Use two variables and assign them a numeric value
  a=10
  b=20
- Print out the results of
  a+b b-a b/a a*b

Solution:

```
a=1
b=2
let c=a+b
let d=b-a
let e=a*b
let f=b/a
echo $c $d $e $f
```

78

Performing Boolean Tests with the Test Command

- The test command's Boolean operators let you combine multiple expressions with AND and OR relationships
- The -a operator combines two expressions and tests a logical AND relationship between them
- The form of the test command with the -a option is:

```
test expression1 -a expression 2
```

- Next screen: use the test command to determine if a directory exists

79

Testing Files with the Test Command

- The test command can be used to determine
  - if a file exists
  - if it has a specified permission or attribute (such as executable, readable, writeable, directory, etc.)

80

Testing Files with the Test Command

Option	Meaning	Example
-e	True if a file exists	test –e file
-r	True if a file exists and is readable	test –r file
-w	True if a file exists and is writeable	test –w file
-x	True if a file exists and is executable	test –x file
-d	True if a file exists and is a directory	test –d file
-f	Tests if a file exists and is a regular file	test –f file
-s	True if a file exists and has a size greater than zero	test –s file
-c	Tests if a file exists and is a character special file (which is a character-oriented device, such as a terminal or printer)	test –c file
-b	Tests if a file exists and is a block special file (which is a block-oriented device, such as a disk or tape drive)	test –b file

Table 7-3: File testing options of test command

81

Logical Operators - Examples

`test –rw fleas`
- not allowed, you cannot combine options like this

`test ! –r flam`
- `!` Negates the expression that follows

`test –r flam –a –w flam`
- `-a` Binary and operator

`test –r flam –o –w flam`
- `-o` Binary or operator

- Parentheses may also be used for grouping

82

Shell Logic Structures

- The four basic logic structures needed for program development are:
  - Sequential logic
  - Decision logic
  - Looping logic
  - Case logic

83

Sequential Logic

- Sequential logic states that commands will be executed in the order in which they appear in the program
- The only break in this sequence comes when a branch instruction changes the flow of execution

84

Decision Logic - if Statement

```
if control command
then
   commands
else
   commands
fi
```

- You can nest a control structure, such as the if statement, inside another control structure
  - The first if statement controls when the second if statement is executed
  - The second if statement is located in the first if statement's else section

85

if then … elif then Statement

```
if control command1
then
   commands
elif control command2
then
   commands
else
   commands
fi
```

86

if Statement Example

```
if [ $day = "Monday" ]
then
  echo First day of the work week.
  echo Search todo for Monday.
  grep -i monday todo
fi
```

87

`if` Statement Example 2

```
a=1
b=2
if [ $a -lt $b ]; then
     echo "$a is less than $b"
  elif [$a -eq $b]; then
     echo "$a is equal to $b"
  else
     echo "$a is greater than $b"
fi
```

88

`if` Statement Logical Examples

- `-a` logical and
  `if [ $a –lt $b –a $b –lt $c ]`
- `-o` logical or
  `if [ $a –lt $b –o $b –lt $c ]`
- `-z` Tests if string length is zero
  `if [ -z $test ]`
- `-n` Tests if string length is greater than zero
  `if [ -n $test ]`

89

Other `if` Capabilities

```
If [ -d $file ]  - is $file a directory
   [ -f $file ]  - is $file a regular file
   [ -r $file ]  - is read permission set
   [ -w $file ]  - is write permission set
   [ -x $file ]  - is execute permission set
   [ -s $file ]  - is the file empty
```

90

Looping Logic

- In looping logic, a control structure (or loop) repeats until some condition exists or some action occurs
- Two looping mechanisms in this section:
  - for loop
  - while loop

91

`for` Loops

- Use the for command for looping through a range of values
  - causes a variable to take on each value in a specified set, one at a time, and perform some action while the variable contains each individual value

```
for i in fly spider frog
do
  echo $i
  echo $i
done
```

92

`for` Loops – Page 2

```
for i in $*
do
  echo $i
done
```

93

`for` Loops – Page 3

- `for i in fly spider frog`
- `for i in $*`
- `for I`
- `For file in *`

94

The `while` Loop

- A different pattern for looping is created using the while statement
- The while statement best illustrates how to set up a loop to test repeatedly for a matching condition
- The while loop tests an expression in a manner similar to the if statement
- As long as the statement inside the brackets is true, the statements inside the do and done statements repeat

95

`while` Loop Syntax

```
while control command
do
  commands
done
```

96

`while` Loop Example

```
echo –n "try to guess my favorite color "
read guess
while [ "$guess" != "red" ];
do
   echo "No, not that one.  Try again. "
   read guess
done
```

97

The `until` Statement

- A looping structure
  - Used to execute a series of commands until a specific condition is true

```
until expression
do
    statements
done
```

```
until [ “$loopcount” –gt 5 ]
do
    echo “$loopcount”
    let loopcount=$loopcount+1
done
```

98

Case Logic
The Case Statement

- The case statement simplifies the selection of a match when you have a list of choices
- It allows your program to perform one of many actions, depending upon the value of a variable
- The two semicolons ;; terminate the action(s) taken after the case matches to the test

99

Case Statement Syntax

```
case value in
  choice1) commands ; ;
  choice2) commands ; ;
  …
esac
```

choice1 and choice2 are labels

100

Case Statement Example

```
read a
case $a in
   1 ) echo "You entered 1" ;;
   2 ) echo "You entered 2" ;;
   3 | 4 ) echo "You entered 3 or 4" ;;
   * ) echo "You entered something else" ;;
esac
```

101

Activity: `if` Statement

```
echo –n "What is your favorite operating
  system? "
read OS_NAME
if [ "$os_NAME" = "UNIX" ]
then
   echo "You will like Linux."
else
   if [ "$OS_NAME" = "Windows" ]
   then
      echo "A great OS for applications."
   else
      echo "You should give Linux a try!"
   fi
fi
```

102

Activity: `for` Loop

```
for USERS in john ellen tom becky jim
do
   echo $USERS
done
```

103

Menu Script Example

```
#!/bin/sh
while true
do
# Display a menu
  echo
  echo "Make a choice from the menu below:"
  echo
  echo "1 Restore Archive
  echo "2 Backup directory"
  echo "3 Quit"
  echo
  # Read the user's selection
  echo "Enter Choice: "
  read CHOICE
  echo $CHOICE
done
```

104

Positional Parameters

Parameter	Meaning / Purpose
$0	Command or name of script
$1 - $9	Command line argument number
$*	All arguments entered on the command line
$#	Number of arguments entered

- Information can be passed into a shell script on the command line in the form of *arguments*.
- These arguments are stored in special variables.
- This was used in the srm script earlier.

105

Exit Status Command

- A command run from the command line or a shell script returns a value to the parent process indicating its success or failure called an *Exit Status*.
- A command defines what a given exit status means. The convention is that a return value of 0 (zero) is a success, and a non-zero value is failure.
- This is how if statements determine which blocks of code to execute.
- The variable `$?` is defined automatically by the shell to hold the exit status of the last command executed.
- Use `exit 0` as the last line of a shell script to indicate successful completion.

106