CSCE 3600
Principles of Systems Programming

Bourne Again Shell (Bash)

University of North Texas

Shell as a User Interface

- A shell is a command interpreter, an interface between a human (or another program) and the OS
  - Runs a program, perhaps the ls program
  - Allows you to edit a command line
  - Can establish alternative sources of input and destinations for output for programs
- It is really just another program

Shell is the interface between end users and the Linux system, similar to the commands in Windows

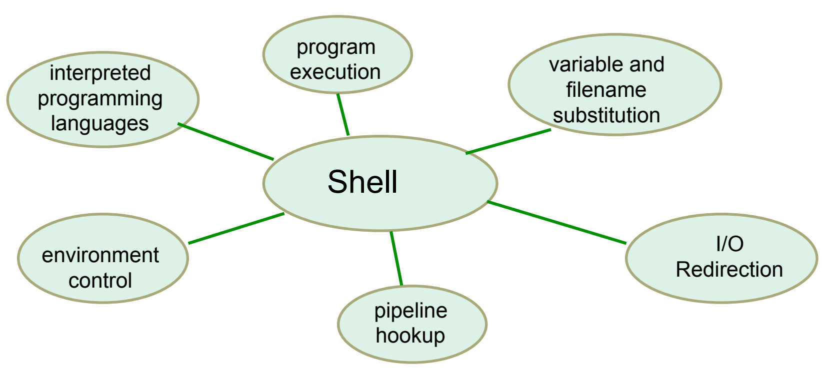
2

Bourne-Again Shell (bash)

- Extension of the Bourne Shell (sh)
  - Check the version:
    - which bash
    - bash --version
- Incorporates many useful features from the Korn shell (ksh) and C shell (csh)
- There are other shell versions:
  - tcsh, zsh, csh, ksh, etc.
- Why shell?
  - For routing jobs, such as system administration, without writing programs

3

Shell's Responsibilities



4

Shell's Responsibilities

- Program execution
  - The shell executes all programs requested by the user
  - Each line interpreted according to format: program_name arguments
  - White space between the program name and the individual arguments is ignored
    - Some commands, like cd, pwd, and echo are built into the shell
    - The rest are utilities that the shell must retrieve from the disk
- Variable and file name substitution
  - The shell permits users to create shell variables that can be assigned values, just as in any of the common programming languages
  - The shell also permits use of "wildcard" characters to generate lists of files to be passed to the chosen command or utility
    - These "wildcards" consist of characters such as the *, ? and [...]
    - The wildcard is replaced by the appropriate files, which are then passed on the command line to the utility

5

Shell's Responsibilities (cont'd)

- I/O Redirection
  - The shell takes care of all redirection of input and output, scanning for the redirection symbols like <, >, and >>
- Pipeline Hookup
  - The shell is also responsible for detecting the use of the pipe symbol |, and connecting the standard output of the preceding command with the standard input of the succeeding command
- Environment Control
  - The shell environment can be customized by each user to include the default home directory, the cursor presented by the shell for prompting and other options to be discussed later
- Interpreted Programming Languages
  - The shell provides a built-in capability for developing fairly complicated programs, or "shell scripts" to automate repetitive tasks
    - It includes variables, arrays, decision making capabilities, looping constructs and arithmetic operations

6

Shell Startup

- The shell is actually the result of the login process
  - Starting off with the kernel's booting processes, key processes involved in starting up the shell include:

  - init replaced by systemd in many Linux distributions
    - Invoked by kernel at end of startup procedure, the init process is responsible for starting up and shutting down the system
  - getty
    - Created by the init process, the getty process is responsible for getting and managing the terminal type (tty)
  - login
    - After communication is established, the login process authenticates the user
    - Once authenticated, the appropriate shell is created

7

Shell Metacharacters

- Some characters have special meaning to the shell
- These are just a few:
  - I/O redirection

    < > |
  - Wildcards

    * ? [ ]
  - Others

    & ; $! \ ( ) space tab newline
- These must be escaped or quoted to inhibit special behavior

8

Shell Substitutions

- Certain characters are interpreted by the various Linux shells as “wildcards” for filenames, also known as metacharacters through globbing
  - * Matches 0 or more of any character
    - ls *.c
  - ? Matches any single character
    - ls file0?.c
  - [...] Matches any single character if it is in list provided
    - ls file[0-9].c
    - ls [a-zA-Z]*
    - ls [!0-9]*

9

Basic Shell Scripting

- A shell script is a file that contains shell commands
  - Data structure: variables
  - Control structure: arithmetic, functions, branches, loops
- Specify shell to execute program
  - Script must begin with #! (pronounced “shebang”) to identify shell to be executed
  - #! /bin/bash
- To run:
  - Make executable: chmod +x script.sh
  - Invoke via: ./script.sh

Typically, use .sh extension to indicate shell script

10

Example "hello" Script

```
#! /bin/bash
echo "You are logged in as ($USER) to machine (`uname -n`)."
echo "This month's calendar is:"
cal
echo "You are currently running the following processes:"
ps u
```

11

Subshells

- A subshell is a new shell created within current shell
  - Each subshell has its own environment
- A subshell can be created by:
  1. Executing the bash command
  2. Starting a background process
  3. Running a shell script
  4. Grouping shell commands within parentheses
     - pwd; (cd /; pwd); pwd
- A subshell cannot change a variable in parent shell
- Subshell variables destroyed on exit

12

Environment Variables

- Bash supports two types of variables:
  - Local variables
  - Environment variables, such as $USER seen earlier
- Environment variables are set by the system and can usually be found using the env command

  env [options] [varname=value] [command]
  - Displays the current environment or modifies the specified variables
  - Specified commands are executed in the new environment

13

The $PATH Variable

- Initially set when the shell is created (login process)
  - Provides a list of available directories where an executable command may be found
- Add new "path" to environment variable

  PATH=$PATH:$HOME/bin

  export PATH
  - This allows us to include another directory in the search path for command
- Directories are separated by colon (:)
- command not found message returned on unsuccessful search

14

Shell Scripting Features

- Full scripting language
  - Conditional statements (if-then-else, case, …)
  - Arithmetic, string, file, environment variables
  - Input (prompt user, command-line arguments, …)
  - Loop statements (for, while, do-while, repeat-until, …)
  - Lists (and, or)
  - Functions
  - Traps
- Small differences among shells
- Bash has a 3000+ line man page

15

Local Variables

- Variable structure format

  varname=value
  - Variable name must begin with alphabetic or underscore character, followed by zero or more alphanumeric or underscore characters
  - Note there should not be any spaces around the '=' sign
  - Variables are case-sensitive
  - Do not use globbing metacharacters, such as ? and *, in your variable names
  - Once defined, use by prefixing $ symbol to variable name
- Example

```
score=100
echo You scored a $score on the exam
```

16

Local Variables

- Example

  cat test1

  echo %$var1%

  ./test1

  %%

  – var1 is equal to NULL

  cat test2

  var2=25

  echo %$var2%

  ./test2

  %25%

  – var2 is equal to 25

17

Exported Variables

- Variables can be exported
  – The value of the variable can be passed to other subshells
- Export command format:

  export [varname]

- Caution 

  – Changing an exported variable in a subshell does *not* change the value in the parent shell

18

Exported Variables (cont'd)

- Example

cat test3

```
var3=2345
echo var3 = $var3
```

./test3

```
var3 = 2345
```

cat test4

```
echo var3 = $var3
```

./test4

```
var3 =
```

var3 is a local variable, so its value in test3 is 2345 while its value in test4 is null

19

Exported Variables (cont'd)

- Example

var3=2345

export var3

cat test3

```
var3=2345
echo var3 = $var3
```

./test3

```
var3 = 2345
```

./test4

```
var3 = 2345
```

var3 is an exported variable, so its value in test3 is 2345 and its value in test4 is also 2345

20

Exported Variables (cont'd)

- Once a variable is exported, it is maintained as an exported variable
  - Each subshell makes its own copy of the variable
  - Unless you use unset to destroy it
- export with no arguments lists the variables that are exported to the user's shell

21

Special Shell Variables

- Arguments can be used to modify script behavior
- Command-line arguments become positional parameters to shell script

Parameter	Meaning
$0	Name of the current shell script
$1-$9	Positional parameters 1 through 9
$#	The number of positional parameters
$*	All positional parameters, "$*" is one string
$@	All positional parameters, "$@" is a set of strings
$?	Return status of most recently executed command
$$	Process id of current process

22

Command Line Arguments Example

```
set tim bill ann fred
     $1   $2   $3   $4
echo $*
tim bill ann fred
echo $#
4
echo $1
tim
echo $3 $4
ann fred
```

The 'set' command can be used to assign values to positional parameters

23

User Input

- Bash allows to prompt for and read in user input
  - The read command allows you to prompt for input and store it in a variable
- Syntax:

  read varname [varname1] [varname2] … [varnameN]
  - or

  read –p "prompt" varname1 [varname2] … [varnameN]
- Input entered by user are assigned to varname1, varname2, etc.
- If more input is entered than there are variables, the remaining input will be assigned to the last variable

24

Quoting Mechanisms

- Quote characters and the backslash character have special meaning in shell scripts
  - ` perform command substitution
  - " allow some variable expansion, but prevent wildcard replacement
  - ' prevent wildcard replacement as well as variable and command substitution
  - \ preserve the literal value of the next character

The general rule is that double quotes still allow expansion of variables within the quotes, and single quotes do not

25

Using Quotes

- Example (single quote):

```
cat city
Austin
Dallas
Ft. Worth
San Antonio
grep 'San Antonio' city
San Antonio
```

26

Using Quotes

- Example (double quote):

```
ex1="'Dallas,' A city in Texas"
echo $ex1
'Dallas,' A city in Texas
ex2='"San Antonio," is also a city in Texas'
echo $ex2
"San Antonio," is also a city in Texas
```

27

Using Quotes

- Example (backslash):

```
echo $ex2
"San Antonio," is also a city in Texas
echo \$ex2
$ex2
```

which is the same as:

```
echo '$'ex2
$ex2
```

28

Using Quotes

- Example (back quote):

```
echo The date and time is: `date`
```

The date and time is: Wed Sep 11 10:05:50 CDT 2019

29

Using Quotes

- Expands *
  ```
  echo *
  ```
- Does not expand *
  ```
  echo "*"
  ```
- Does not expand *
  ```
  echo '*'
  ```
- Performs command substitution and expands $HOME and *
  ```
  echo "$HOME directory files `ls *`"
  ```
- Performs command substitution and expands *, but not $HOME
  ```
  echo "\$HOME directory files `ls *`"
  ```
- No command substitution nor expands $HOME and *
  ```
  echo '$HOME directory files `ls *`'
  ```

30

Arithmetic Evaluation

- let statement can be used to perform arithmetic operations
- Available operators are: +, –, *, /, %

```
let X=10+2*7
echo $X
let Y=X+2*4
echo $Y
```

- An arithmetic expression can be evaluated by:

  $[expression] or $((expression))

```
echo "$((123+20))"
VALORE=$[123+20]
echo "$[123*$VALORE]"
```


Condition Expressions

- Conditionals let us decide whether to perform an action or not by evaluating an expression
- Condition expressions must be enclosed in either
  - Single brackets […] are built-in in bash as an alias for the test command
    - Conditions can also be evaluated without the single brackets using the test command with the general format

      if test condition
  - Double brackets [[…]] is a bash keyword and is much more capable than single brackets, though may not be supported by all versions (bash, zsh, and ksh support it)
    - Also allows for more C-like syntax with relational operators

Condition Expressions

- The most basic form of condition expressions is:

```
if [ expression ]
then
   statement(s)
elif [ expression ]
then
   statement(s)
else
   statement(s)
fi
```

- The elif (else if) and else sections are optional
- Put spaces after [and before], and around the operators and operands

The test File Operator

- The test file operator returns true if:

–d file	file is a directory
–f file	file is an ordinary file, and exists
–r file	file is readable by the process
–s file	file is not empty
–w file	file is writable by the process
–x file	file is executable
–G file	file is owned by the group user belongs to
–O file	file is owned by the user
–u file	set_user_id bit is set
–g file	set_group_id bit is set

The test Command

- Example (using the test command)

```
if test –w "$1"
then
      echo "file $1 is writeable"
fi
```

- More examples (using single brackets)

  [–f /usr/train1/file1]
  - If file1 exists and is an ordinary file

  [–r /usr/train2/file1]
  - If file1 exists and is readable by this process

  [–s /usr/train3/file1]
  - If file1 exists and is not empty

35

Relational Operators

Meaning	Numeric	String
Greater than	-gt	
Greater than or equal	-ge	
Less than	-lt	
Less than or equal	-le	
Equal	-eq	= or ==
Not equal	-ne	!=
str1 is less than str2		str1 < str2
str1 is greater str2		str1 > str2
String length is greater than zero		-n str
String length is zero		-z str

36

Logical Operators

- Compound expressions formed with logical operators

! expression	logical negation
expression –a expression	logical and
expression –o expression	logical or

- Examples

```
[ ! –f /usr/train1/file1 ]
[ –f /usr/train1/file1 –a –r /usr/train1/file1 ]
[ –n "$var1" –o –r /usr/train2/file1 ]
```

The && (and) and || (or) logical operators may also be used, but must be enclosed in [[…]]

37

Example if … Statement

```
# The following three if-conditions produce the same result
#! /bin/bash
# DOUBLE SQUARE BRACKETS
read –p "Do you want to continue? " reply
if [[ $reply = "y" ]]; then
    echo "You entered " $reply
fi
# SINGLE SQUARE BRACKETS
read –p "Do you want to continue? " reply
if [ $reply = "y" ]; then
    echo "You entered " $reply
fi
# "TEST" COMMAND
read –p "Do you want to continue? " reply
if test $reply = "y"; then
    echo "You entered " $reply
fi
```

38

The case Statement

- Use the case statement for a decision that is based on multiple choices
- Syntax:

```
case value in
      pattern1)    statement-list1
      ;;
      pattern2)    statement-list2
      ;;
      …
      patternN)    statement-listN
      ;;
esac
```

The first pattern to match value will be executed

- Patterns may also contain meta-characters, such as *, ?, […], and character classes
- Multiple patterns are also supported through the use of the | operator

39

The case Statement Example 1

```
#! /bin/bash
echo "Enter Y to see all the files including hidden files"
echo "Enter N to see all non-hidden files"
echo "Enter q to quit"

read –p "Enter your choice: " reply

case $reply in
    Y|YES)    echo "Displaying all (really…) files"
              ls –a ;;
    N|NO)     echo "Displaying all non-hidden files"
              ls ;;
    *)        echo "Invalid choice! "; exit 1 ;;
esac
```

40

The case Statement Example 2

```
#! /bin/bash
ChildRate=3
AdultRate=10
SeniorRate=7
read –p "Enter your age: " age
case $age in
    [1-9]|[1][0-2])             # child, age 1 - 12
            echo "Your rate is" '$' "$ChildRate.00" ;;
    [1][3-9]|[2-5][0-9])        # adult, age 13 - 59
            echo "Your rate is" '$' "$AdultRate.00" ;;
    [6-9][0-9])                 # senior, age 60+
            echo "Your rate is" '$' "$SeniorRate.00" ;;
esac
```

41

The while Loop

- Bash supports the while loop to execute statements as long as the condition holds true with the following format:

```
while [ expression ]
do
    statement1
    statement2
    …
    statementN
done
```

42

Using the while Loop Example 1

```
#! /bin/bash

index=1
while [ $index -le 10 ]
do
    echo loop: $index
    let index=$index+1
done
```


Using the while Loop Example 2

```
#! /bin/bash

Again="Y"
while [ $Again = "Y" ]; do
    ps u
    read –p "Do you want to continue? (Y/N) " reply
    Again=`echo $reply | tr [:lower:] [:upper:]`
done
echo "done"
```

The until Loop

- Bash supports the until loop to execute statements as long as the condition holds false with the following format:

```
until [ expression ]
do
    statement1
    statement2
    …
    statementN
done
```


Using the until Loop Example 1

```
#! /bin/bash

COUNTER=20
until [ $COUNTER –lt 10 ]
do
    echo $COUNTER
    let COUNTER–=1
done
```

Using the until Loop Example 2

```
#! /bin/bash

stop="N"
until [ $stop = "Y" ]; do
    ps
    read –p "Do you want to stop? (Y/N) " reply
    stop=`echo $reply | tr [:lower:] [:upper:]`
done
echo "done"
```

47

The for Loop

- Although the for loop is supported in the traditional sense (i.e., when the number of iterations is known), we look at using the for loop to iterate over a list of arguments with the following format:

```
for varname in arg1 arg2 … argN
do
    statement1
    statement2
    …
    statementN
done
```

The for loop is a little different from other programming languages as it basically lets you iterate over a series of "words" within a string

48

Using the for Loop Example 1

```
#! /bin/bash
for i in 7 9 2 3 4 5
do
    echo $i
done
```

- The simplest form will iterate over all command line arguments:

```
#! /bin/bash
for parm
do
   echo $parm
done
```


Using the for Loop Example 2

```
#! /bin/bash
for quizNum in 1 2 3 4 5
do
    read -p "Enter quiz #$quizNum: " score
    let sum=$sum+$score
done
let quizAvg=$sum/5
echo "Average quiz grade: $quizAvg"
```

- Note this computation results in integer values only!
- If we wanted accurate floating-point results, we must use a precision calculator, such as bc

A C-like for Loop

- An alternative form of the for structure is:

```
for (( expression1 ; expression2 ; expression3 ))
do
    statement1
    statement2
    …
    statementN
done
```

```
#!/bin/bash
echo –n "Enter a number: "; read x
let sum=0
for (( i=1 ; $i<=$x ; i=$i+1 ))
do
   let "sum = $sum + $i"
done
echo "sum of first $x numbers is: $sum"
```

51

The select Command

- select command constructs simple menu from a list
  - Allows user to enter a number instead of a string value
  - User enters sequence number corresponding to the argument in the list

```
select value in LIST
do
    statement(s)
done
```

52

Using select Example

```
#! /bin/bash

select var in alpha beta gamma
do
    echo $var
done
```

```
1) alpha
2) beta
3) gamma
#? 2
beta
#? 4

#? 1
alpha
```

53

The select Command in Detail

- PS3 is select sub-prompt prompt statement
  - It is the prompt used by “select” inside shell script
  - Prompt statements: PS1, PS2, PS3, PS4, PROMPT_COMMAND
- $REPLY is user input (the number)

```
#! /bin/bash
PS3="select entry or ^D: "
select var in alpha beta
do
    echo "$REPLY=$var"
done
```

```
Output:
select ...
1) alpha
2) beta
? 2
2 = beta
? 1
1 = alpha
```

54

break and continue

- Interrupts the for, while, or until loop
- The break statement
  - Transfers control to the statement after the done statement
  - Terminates execution of the loop
- The continue statement
  - Transfers control to the done statement
  - Skips the test statements for the current iteration
  - Continues execution of the loop

55

The break Command

```
while [ condition ]
do
    statement1
    break
    statementN
done
echo "done"
```

This iteration is over and there are no more iterations

56

The continue Command

```
while [ condition ]
do
    statement1
    continue
    statementN
done
echo "done"
```

This iteration is over;
do the next iteration

continue and break Example

```
for index in 1 2 3 4 5 6 7 8 9 20
do
    if [ $index –le 3 ]; then
        echo "continue"
        continue
    fi
    echo $index
    if [ $index –ge 8 ]; then
        echo "break"
        break
    fi
done
```

Shell Functions

- A shell function is similar to a shell script
  - Stores a series of commands for execution later
  - Shell stores functions in memory
  - Shell executes a shell function in the same shell that called it (not a subshell)
- Must be defined before they can be referenced
  - Usually placed at the beginning of the script

```
function-name()
{
    statements
}
```

59

Function Example 1

```
#! /bin/bash

funky()
{
    # This is a simple function
    echo "This is a funky function."
    echo "Now exiting funky function."
}

# declaration must precede call:
funky
```

60

Function Example 2

```
#! /bin/bash
fun()
{     # A somewhat more complex function.
      JUST_A_SECOND=1
      let i=0
      REPEATS=30
      echo "And now the fun really begins."
      while [ $i –lt $REPEATS ]
      do
            echo "-------FUNCTIONS are fun-------->"
            sleep $JUST_A_SECOND
            let i+=1
      done
}
fun
```

61

Function Parameters

- Functions do not need to define formal parameters
  - They are supported by positional parameters (i.e., $1, $2, …) for each argument passed to the function

    $# reflects number of parameters

    $0 still contains name of script (not name of function)
- Functions invoked with or without parameters
  - But no parentheses are needed in the function call

62

Function with Parameter Example 1

```
#! /bin/bash
testfile()
{
    if [ $# –gt 0 ]; then
        if [[ –f $1 && –r $1 ]]; then
            echo $1 is a readable file
        else
            echo $1 is not a readable file
        fi
    fi
}
testfile .
testfile funtest
```

63

Function with Parameter Example 2

```
#! /bin/bash
checkfile()
{
    for file
    do
        if [ -f "$file" ]; then
            echo "$file is a file"
        else
            if [ -d "$file" ]; then
                echo "$file is a directory"
            fi
        fi
    done
}
checkfile . funtest
```

64

Local Variables in Functions

- Variables defined within functions are global
  - Their values are known throughout the entire shell program
- Keyword "local" inside a function definition makes referenced variables "local" to that function

Function Variables Example

```
#! /bin/bash
global="pretty good variable"
checkvar()
{
    local inside="not so good variable"
    echo $global
    echo $inside
    global="better variable"
}

echo $global
checkvar
echo $global
echo $inside
```

Using Arrays

- Arrays up to 1024 elements are created when used in one of two subscript forms:

  pet[0]=dog
  pet[1]=cat or pet=([0]="dog" [1]="cat" [2]="fish") or pet=(dog cat fish)
  pet[2]=fish
- To extract a value, use ${arrayname[i]}

  echo ${pet[0]}
- To extract all elements, use ${arrayname[*]}
  - You can combine arrays with loops using a for loop

```
for x in ${arrayname[*]}
do
    …
done
```

${#arrayname[*]} for number of elements in array

67

String Manipulation

- Bash supports a number of string manipulation operations

  ${#string} gives the string length

  ${string:pos} extracts substring from $string at $pos

  ${string:pos:len} extracts $len characters from $string at $pos
- Example

```
str=0123456789
echo ${#str}
echo ${str:6}
echo ${str:6:2}
```

68

Signal Handling

- Linux allows you to send a signal to any process
  - Any Linux process can be interrupted by a signal
    - Ctrl-C (^C) typed via keyboard either stops a program from running or terminates bash
  - Signal end of input with EOF signal, Ctrl-D (^D)
    - Pressing Ctrl-D at the shell causes the shell to exit
    - What if you don't want to exit the shell?

      ignoreeof=1 bash
- To see a list of supported signals in Linux

  kill –l

69

Signal Handling

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

- List your processes with ps u or ps –u userid

–1 = hangup	kill –SIGHUP 1234
–2 = interrupt with ^C	kill –2 1235
no argument = terminate	kill 1236
–9 = kill (force terminate)	kill –9 1237

70

Handling Signals

- Default action for most signals is to end process
- Bash allows you to install a custom signal handler
- Syntax:

  trap ['handler command'] signal1 [signal2] … [signalN]

  – The 'handler command' can be a single command, such as echo, or it can be a function call
  – trap followed only by signal number resets signal handler
- Example

  trap 'echo do not hangup' 1 2

Trap Hangup Example

```
#! /bin/bash
# kill –1 won't kill this process
# kill –2 will (may not always work)

trap 'echo do not hang up' 1

while true
do
    echo "try to hang up"
    sleep 1
done
```

Trap Multiple Signals Example

```
#! /bin/bash
# plain kill or kill –9 will kill this

trap 'echo 1' 1
trap 'echo 2' 2

while true
do
    echo –n .
    sleep 1
done
```


Restoring Default Handlers

- Use this to run a signal handler once only

```
#! /bin/bash
trap 'echo SIGHUP will not work' 1
trap 'suppressonce' 2
suppressonce()
{
    echo "SIGINT suppressed"
    trap 2  # reset it (may not always work)
}
while true
do
    echo -n "."
    sleep 1
done
```

Be aware that when using custom signal handlers, its behavior can be somewhat flaky and not work every time

Debugging Shell Scripts

- Bash provides two useful options for debugging:
  - echo
    - Use explicit output statements to trace execution
  - set with options to allow flow of execution
    - –v option prints each line as it is read
    - –x option displays the command and its arguments
    - –n option checks for syntax errors
  - Options can be turned on or off
    - To turn on the option: set –xv
    - To turn off the option: set +xv
  - Options can also be set via she-bang line
    - #! /bin/bash –xv

May also execute in debug mode at command line:
bash –x scriptname.sh

75

Debugging Example

```
#!/bin/bash –x
echo –n "Enter a number: "; read x
let sum=0
for (( i=1 ; $i<$x ; i=$i+1 ))
do
    let "sum = $sum + $i"
done
echo "the sum of the first $x numbers is: $sum"
```

76