Introduction to Bash Scripting

Dr. Vimal Baghel
Assistant Professor
SCSET, BU

Shell Scripts (1)

- Basically, a shell script is a text file with Unix commands in it.
- Shell scripts usually begin with a #! and a shell name
  - For example: #!/bin/sh
  - If they do not, the user's current shell will be used

abc.sh

- Any Unix command can go in a shell script
  - Commands are executed in order or in the flow determined by control statements.
- Different shells have different control structures
  - The #! line is very important
  - We will write shell scripts with the Bourne again shell (bash)

gedit abc

Bash scripting features

- **Programming features of the UNIX/LINUX shell:**
  - ***Shell variables*: Your scripts often need to keep values in memory for later use. Shell variables are symbolic names that can access values stored in memory**
  - ***Operators*: Shell scripts support many operators, including those for performing mathematical operations**
  - ***Logic structures*: Shell scripts support sequential logic (for performing a series of commands), decision logic (for branching from one point in a script to another), looping logic (for repeating a command several times), and case logic (for choosing an action from several possible alternatives)**

Steps in Writing a Shell Script

- Write a script file using vi:
  - The first line identifies the file as a **`bash`** script.

    `#!/bin/bash`
  - Comments begin with a **#** and end at the end of the line.
- give the user (and others, if (s)he wishes) permission to execute it.
  - chmod +x filename

chmod ug +x file

chmod 001 file

- Run from local dir
  - ./filename
- Run with a trace – echo commands after expansion
  - bash –x ./filename

Shell Scripts (2)

```
$ ps -p $$
```

The output is as follows:

```
  PID TTY          TIME CMD
12578 pts/4    00:00:00 bash
```

- Why write shell scripts?
  - To avoid repetition:
    - If you do a sequence of steps with standard Unix commands over and over, why not do it all with just one command?
  - To automate difficult tasks:
    - Many commands have subtle and difficult options that you don't want to figure out or remember every time.

Writing your First Bash Shell Script

- Find out where is your Bash interpreter located
  - $ which bash
  - /usr/bin/bash
- Open our favorite text editor and create a file hello.sh
  - $ gedit hello.sh
- Write the script as:
  - #!/usr/bin/bash
  - # declare STRING variable
  - STRING="Hello World"
  - # print variable on a screen
  - echo $STRING

Shebang

$ chmod u+x hello.sh

Execution of your first bash script

- Make the file hello.sh executable as:
  - `$ chmod +x hello.sh`
- Execute your first bash script as:
  - `$ ./hello.sh` **OR** `$ bash hello.sh`
  - `Output: Hello World`
- You are done!

A Simple Example (1)

- $ tr abcdefghijklmnopqrstuvwxyz \thequickbrownfxjmpsvalzydg < file1 > file2
  - “encrypts” file1 into file2
- Record this command into shell script files as:
  - myencrypt

    #!/bin/sh

    tr abcdefghijklmnopqrstuvwxyz \thequickbrownfxjmpsvalzydg
  - mydecrypt

    #!/bin/sh

    tr thequickbrownfxjmpsvalzydg \abcdefghijklmnopqrstuvwxyz

A Simple Example (2)

- chmod the files to be executable; otherwise, you couldn't run the scripts

```
$ chmod u+x myencrypt mydecrypt
```

- Run them as normal commands:

```
$ ./myencrypt < file1 > file2
$ ./mydecrypt < file2 > file3
$  diff file1 file3
```

Remember: This is needed when "." is not in the path

Bourne Shell Variables

- Remember: Bash shell variables are different from variables in csh and tcsh!
  - Examples in sh:

```
PATH=$PATH:$HOME/bin
HA=$1
PHRASE="House on the hill"
export PHRASE
```

Note: no space around =

Make PHRASE an environment variable

Assigning Command Output to a Variable

- Using backquotes, we can assign the output of a command to a variable:

```
#!/usr/bin/bash
files=`ls`
echo $files
```

- Very useful in numerical computation:

```
#!/usr/bin/sh
value=`expr 12345 + 54321`
echo $value
```

Using expr for Calculations

- Variables as arguments:

```
$ count=5
$ count=`expr $count + 1`
$ echo $count
6
```

  - Variables are replaced with their values by the shell!
- expr supports the following operators:
  - arithmetic operators: +,-,*,/,%
  - comparison operators: <, <=, ==, !=, >=, >
  - boolean/logical operators: &, |
  - parentheses: (,)
  - precedence is the same as C, Java

Variables

- Create a variable
  - Variablename=value (no spaces, no $)
  - read variablename (no $)
- Access a variable's value
  - $variablename
- Set a variable
  - Variablename=value (no spaces, no $ before variablename)

Variables

- **Variables are symbolic names that represent values stored in memory**
- **Three different types of variables**
  - Global Variables**: Environment and configuration variables, capitalized, such as HOME, PATH, SHELL, USERNAME, and PWD.**

    When you login, there will be many global System variables that are already defined. These can be freely referenced and used in your shell scripts.
  - Local Variables

    Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell.
  - Special Variables

    Reserved for OS, shell programming, etc. such as positional parameters $0, $1 …

Variable Scope & Processes

- Variables are shared only with their own process, unless exported
  - x=Hi – define x in current process
  - sh – launch a new process
  - echo $x – cannot see x from parent process
  - x=bye
  - <ctrl d> -- exit new process
  - echo $x -- see x in old process did not change
  - demoShare – cannot see x
  - . demoShare – run with dot space runs in current shell
  - export x – exports the variable to make available to its children
  - demoShare – now it can see x

Positional Parameters

When a shell script is invoked with a set of command line parameters each of these parameters are copied into special variables that can be accessed.

Positional Parameter	**What It References**
`$0`	References the name of the script
`$#`	Holds the value of the number of positional parameters
`$*`	Lists all the positional parameters
`$@`	Same as $*, except when enclosed in double quotes
`"$*"`	Expands to a single argument (e.g., "**$1 $2 $3**")
`"$@"`	Expands to separate arguments (e.g., "**$1**" "**$2**""**$3**")
`$1 .. ${10}`	References individual positional parameters
`set`	Command to reset the script arguments

Positional Parameters

- $0 This variable that contains the name of the script
- $1, $2, ….. $n 1^{st}, 2^{nd} 3^{rd} command line parameter
- $# Number of command line parameters
- $$ process ID of the shell
- $@ same as $* but as a list one at a time (see for loops later)
- $? Return code 'exit code' of the last command
- Shift command: This shell command shifts the positional parameters by one towards the beginning and drops $1 from the list. After a shift $2 becomes $1 , and so on … It is a useful command for processing the input parameters one at a time.

Example:

$0 $1 $2 $3 $4 $5

Invoke : ./myscript one two buckle my shoe

During the execution of myscript variables $1 $2 $3 $4 and $5 will contain the values *one*, *two*, *buckle*, *my*, *shoe* respectively.

Environment Variables

- set | more – shows all the environment variables that exist
- Change
  - PS1='\u>'
  - PATH=$PATH:/home/pe16132/bin1
  - IFS=':'
  - IFS is **Internal Field Separator**

$* and $@

- **$*** and **$@** can be used as part of the list in a for loop or can be used as par of it.
- When expanded **$@** and **$*** are the same unless enclosed in double quotes.
  - **$*** is evaluated to a single string while **$@** is evaluated to a list of separate word.

Shell Logic Structures & Control

The four basic logic structures needed for program development are:

- **Sequential logic:** to execute commands in the order in which they appear in the program
- **Decision logic:** to execute commands only if a certain condition is satisfied
- **Looping logic:** to repeat a series of commands for a given number of times
- **Case logic:** to replace “if then/else if/else” statements when making numerous comparisons

Conditionals

- Conditionals are used to "test" something.
  - In Java or C, they test whether a Boolean variable is true or false.
  - In a bash script, the only thing you can test is whether a command is "successful"
- Every well-behaved command returns a return code.
  - 0 if it was successful
  - Non-zero if it was unsuccessful (actually 1..255)
  - This is different from true/false conditions in C.

The if Statement

- Simple form:

```
if decision_command_1
then
    command_set_1
fi
```

- ✓ grep returns 0 if it finds something
- ✓ returns non-zero otherwise

- Example:

```
if  grep unix myfile >/dev/null
then
  echo "It's there"
fi
```

- ✓ redirect to /dev/null so that "intermediate" results do not get printed

if and else

```
if grep “LINUX" myfile >/dev/null
then
  echo LINUX occurs in myfile
else
   echo  No!
  echo LINUX does not occur in myfile
fi
```

if and elif

```
if grep " LINUX " myfile >/dev/null
then
  echo " LINUX occurs in file"
elif grep "DOS" myfile >/dev/null
then
  echo " LINUX does not occur, but DOS does"
else
  echo "Nobody is there"
fi
```

Use of Semicolons

- Instead of being on separate lines, statements can be separated by a semicolon (;)
  - For example:

    ```
    if grep " LINUX " myfile; then echo "Got it"; fi
    ```

  - This actually works anywhere in the shell.

```
$ cwd=`pwd`; cd $HOME; ls; cd $cwd
```

Use of Colon

- Sometimes it is useful to have a command which does “nothing”.
- The : (colon) command in LINUX does nothing

```
#!/bin/sh
if grep LINUX myfile
then
  :
else
  echo "Sorry, LINUX was not found"
fi
```

Test command

String and numeric comparisons used with test or [[ ]] which is an alias for test and also [ ] which is another acceptable syntax

- string1 = string2 True if strings are identical
- String1 == string2 ...ditto....
- string1 !=string2 True if strings are not identical
- string Return 0 exit status (=true) if string is not null
- -n string Return 0 exit status (=true) if string is not null
- -z string Return 0 exit status (=true) if string is null

- int1 -eq int2 Test identity
- int1 -ne int2 Test inequality
- int1 -lt int2 Less than
- int1 -gt int2 Greater than
- int1 -le int2 Less than or equal
- int1 -ge int2 Greater than or equal

The test Command – File Enquiry Options

-d file	Test if file is a directory
-f file	Test if file is not a directory
-s file	Test if the file has non zero length
-r file	Test if the file is readable
-w file	Test if the file is writable
-x file	Test if the file is executable
-o file	Test if the file is owned by the user
-e file	Test if the file exists
-z file	Test if the file has zero length

All these conditions return true if satisfied and false otherwise.

- test –f file does file exist and is not a directory?
- test -d file does file exist and is a directory?
- test –x file does file exist and is executable?
- test –s file does file exist and is longer than 0 bytes?

```
#!/bin/sh
count=0
for i in *; do
    if test –x $i; then
            count=`expr $count + 1`
    fi
done
echo Total of $count files executable.
```

The test Command – String Tests

- test –z string   is string of length 0?
- test string1 = string2   does string1 equal string2?
- test string1 != string2   not equal?

Example:

```
if test -z $REMOTEHOST
then
    :
else
    DISPLAY="$REMOTEHOST:0"
    export DISPLAY
fi
```

The test Command – Integer Tests

- Integers can also be compared:
  - Use -eq, -ne, -lt, -le, -gt, -ge
- For example:

```
#!/bin/sh
smallest=10000
for i in 5 8 19 8 7 3; do
  if test $i -lt $smallest; then
     smallest=$i
  fi
done
echo $smallest
```

Use of [ ]

- The test program has an alias as [ ]
  - Each bracket must be surrounded by spaces!
  - This is supposed to be a bit easier to read.
- For example:

```
#!/bin/sh
smallest=10000
for i in 5 8 19 8 7 3; do
    if [ $i -lt $smallest ] ; then
        smallest=$i
   fi
done
echo $smallest
```

Combining tests with logical operators || (or) and && (and)

- Syntax: if cond1 && cond2 || cond3 …

- An alternative form is to use a compound statement using the –a and –o keywords, i.e.

  if cond1 –a cond22 –o cond3 …

  Where cond1,2,3 .. Are either commands returning a a value or test conditions of the form [ ] or test …

Examples:

```
if  date | grep "Fri"  &&  `date +'%H'` -gt 17
then
        echo "It's Friday, it's home time!!!"
fi

if [ "$a" –lt 0 –o "$a" –gt 100 ]     # note the spaces around ] and [
then
        echo " limits exceeded"
fi
```

Decision Logic

- **Simple Example1**

```
#!/bin/sh
if [ "$#" -ne 2 ] then
        echo $0 needs two parameters!
        echo You are inputting $# parameters.
  else
        par1=$1
        par2=$2
  fi
  echo $par1
  echo $par2
```

Simple Example2:

```
#! /bin/sh
#  number is positive, zero or negative
echo -e "enter a number:\c"
read number
if [ "$number" -lt 0 ]
        then
        echo "negative"
elif [ "$number" -eq 0 ]
        then
        echo zero
else
        echo positive
fi
```

Loops

- Loop is a block of code that is repeated a number of times.
- The repeating is performed either a pre-determined number of times determined by a list of items in the loop count (for loops) or until a particular condition is satisfied (while and until loops)
- To provide flexibility to the loop constructs there are also two statements namely break and continue are provided.

for loops

Syntax:

```
for  arg in list
do
    command(s)
  ...
done
```

Where the value of the variable ***arg*** is set to the values provided in the list one at a time and the block of statements executed. This is repeated until the list is exhausted.

Example:

```
for i in 3 2 5 7
do
      echo " $i times 5 is  $((  $i  * 5 ))  "
done
```

for Loops

- for loops allow the repetition of a command for a specific set of values
  - Syntax:

```
for var in value1 value2 ...
do
  command_set
done
```

  - command_set is executed with each value of var (value1, value2, ...) in sequence

for Loop Example (1)

```
#!/bin/bash
# timestable – print out a multiplication table
for i in 1 2 3
do
   for j in 1 2 3
   do
      value=`expr $i \* $j`
      echo -n "$value "
   done
  echo
done
```

$i

for Loop Example (2)

```
#!/bin/bash
# file-poke – tell us stuff about files
files=`ls`
for i in $files
do
    echo -n "$i "
    grep $i $i
done
```

- Find filenames in files in current directory

for Loop Example (3)

```
#!/bin/bash
# file-poke – tell us stuff about files
for i in *; do
    echo -n "$i "
    grep $i $i
done
```

- Same as previous slide, only a little more condensed.

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

while loops

Syntax:

```
while this_command_execute_successfully
  do
        this command
        and this command
  done
```

EXAMPLE:

```
while test "$i" -gt 0      # can also be  while  [ $i > 0 ]
do
            i=`expr $i - 1`
done
```

The while Loop

- While loops repeat statements as long as the next Unix command is successful.
- For example:

```
#!/bin/sh
i=1
sum=0
while [ $i -le 100 ]; do
   sum=`expr $sum + $i`
   i=`expr $i + 1`
done
echo The sum is $sum.
```

Looping Logic: Examples

```
#!/bin/sh
for person in Bob Amit July Gaurav
do
   echo Hello $person
done
```

Output:

```
Hello Bob
Hello Susan
Hello Joe
Hello Gerry
```

- **Adding integers from 1 to 10**

```
#!/bin/sh
i=1
sum=0
while [ “$i” -le 10 ]
  do
  echo Adding $i into the sum.
  sum=`expr $sum + $i `
          i=`expr $i + 1 `
  done
echo The sum is $sum.
```

until loops

The syntax and usage is almost identical to the while-loops.

Except that the block is executed until the test condition is satisfied, which is the opposite of the effect of test condition in while loops.

Note: You can think of *until* as equivalent to *not_while*

Syntax:

```
until test
do
  commands ….
done
```

The until Loop

- Until loops repeat statements until the next Unix command is successful.
- For example:

```
#!/bin/sh
x=1
until [ $x -gt 3 ]; do
  echo x = $x
  x=`expr $x + 1`
done
```

Switch/Case Logic

- **The switch logic structure simplifies the selection of a match when you have a list of choices**

- **It allows our program to perform one of many actions, depending upon the value of a variable**

Case statements

- The case structure compares a string ‘usually contained in a variable’ to one or more patterns and executes a block of code associated with the matching pattern.
- Matching-tests start with the first pattern and the subsequent patterns are tested only if no match is not found so far.

```
case argument in
   pattern 1) execute this command
              and this
              and this;;
   pattern 2) execute this command
              and this
              and this;;
   esac
```

Command Line Arguments (1)

- Shell scripts would not be very useful if we could not pass arguments to them on the command line
- Shell script arguments are “numbered” from left to right
  - $1 - first argument after command
  - $2 - second argument after command
  - ... up to $9
  - They are called “positional parameters”.

Command Line Arguments (2)

- Example: get a particular line of a file
  - Write a command with the format:

```
getlineno linenumber filename
#!/bin/sh
head -$1 $2 | tail -1
```

- Other variables related to arguments:
  - $0 name of the command running
  - $* All the arguments (even if there are more than 9)
  - $# the number of arguments

Command Line Arguments (3)

- Example: print the oldest files in a directory

```
#! /bin/sh
# oldest -- examine the oldest parts of a directory
HOWMANY=$1
shift
ls -lt $* | tail +2 | tail $HOWMANY
```

- The shift command shifts all the arguments to the left
  - $1 = $2, $2 =$3, $3 = $4, ...
  - $1 is lost (but we have saved it in $HOWMANY)
  - The value of $# is changed ($# - 1)
  - **useful when there are more than 9 arguments**
- The “tail +2” command removes the first line.

More on Bourne Shell Variables (1)

- There are three basic types of variables in a shell script:
  - Positional variables ...
    - $1, $2, $3, ..., $9
  - Keyword variables ...
    - Like $PATH, $HOWMANY, and anything else we may define.
  - Special variables ...

More on Bourne Shell Variables (2)

- Special variables:
  - $*, $#   -- all the arguments, the number of the arguments
  - $$ -- the process id of the current shell
  - $?   -- return value of last foreground process to finish

    -- more on this one later
  - There are others you can find out about with man sh

Reading Variables From Standard Input (1)

- The read command reads one line of input from the terminal and assigns it to variables give as arguments

- Syntax: `read var1 var2 var3 ...`
  - Action: reads a line of input from standard input
  - Assign first word to var1, second word to var2, ...
  - The last variable gets any excess words on the line.

Reading Variables from Standard Input (2)

- Example:

```
% read X Y Z
Here are some words as input
% echo $X
Here
% echo $Y
are
% echo $Z
some words as input
```

The case Statement

- The case statement supports multiway branching based on the value of a single string.
- General form:

```
case string in
 pattern1)
  command_set_11
  ;;
 pattern2)
  command_set_2
  ;;
 …
esac
```

case Example

```
#!/bin/sh
echo -n 'Choose command [1-4] > '
read reply
echo
case $reply in
 "1")
   date
   ;;
 "2"|"3")
   pwd
   ;;
 "4")
   ls
   ;;
 *)
   echo Illegal choice!
   ;;
esac
```

Use the pipe symbol “|” as a logical or between several choices.

Provide a default case when no other cases are matched.

Redirection in bash Shell Scripts (1)

- Standard input is redirected the same (<).
- Standard output can be redirected the same (>).
  - Can also be directed using the notation 1>
  - For example: cat x 1> ls.txt (only stdout)
- Standard error is redirected using the notation 2>
  - For example: cat x y 1> stdout.txt 2> stderr.txt
- Standard output and standard error can be redirected to the same file using the notation 2>&1
  - For example: cat x y > xy.txt 2>&1
- Standard output and standard error can be piped to the same command using similar notation
  - For example: cat x y 2>&1 | grep text

Redirection in bash Shell Scripts (2)

- Shell scripts can also supply standard input to commands from text embedded in the script itself.
- General form: command << word
  - Standard input for command follows this line up to, but not including, the line beginning with word.
- Example:

```
#!/bin/sh
grep 'hello' << EOF
This is some sample text.
Here is a line with hello in it
Here is another line with hello.
No more lines with that word.
EOF
```

Only these two lines will be matched and displayed.

A Shell Script Example (1)

- Suppose we have a file called marks.txt containing the following student grades:

```
091286899  90  H. White
197920499  80  J. Brown
899268899  75  A. Green
……
```

- We want to calculate some statistics on the grades in this file.

A Shell Script Example (2)

```
#!/bin/sh
sum=0; countfail=0; count=0;
while read studentnum grade name; do
   sum=`expr $sum + $grade`
   count=`expr $count + 1`
   if [ $grade -lt 50 ]; then
         countfail=`expr $countfail + 1`
   fi
done
echo The average is `expr $sum / $count`.
echo $countfail students failed.
```

A Shell Script Example (3)

- Suppose the previous shell script was saved in a file called statistics.
- How could we execute it?
- As usual, in several ways ...
  - % cat marks.txt | statistics
  - % statistics < marks.txt
- We could also just execute statistics and provide marks through standard input.

Quote Characters

- There are three different quote characters with different behaviour. These are:

  - “**double quote:**, *weak quote.* If a string is enclosed in “ ” the references to variables (i.e *$variable*) are replaced by their values. Also back-quote and escape \ characters are treated specially.

  - ‘**single quote:**, *strong quote*. Everything inside single quotes are taken literally, nothing is treated as special.

  - `**back quote**: A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the string.

    Example: `echo “Today is:” `date``

Array/list

- **To create lists (array) – round bracket**
  $ set Y = (LSP 123 CSET213)

- **To set a list element – square bracket**
  $ set Y[2] = HUSKER

- **To view a list element:**
  $ echo $Y[2]

a[0] = 1
a[1] = 2
a[2] = 3

Example:

```
#!/bin/sh
a=(1 2 3)
echo ${a[*]}
echo ${a[0]}
```

Results: 1 2 3
1

Functions

- Functions are a way of grouping together commands so that they can later be executed via a single reference to their name.
- If the same set of instructions have to be repeated in more than one part of the code, this will save a lot of coding and also reduce possibility of typing errors.

SYNTAX:

```
functionname()
 {
    block of commands
 }
```

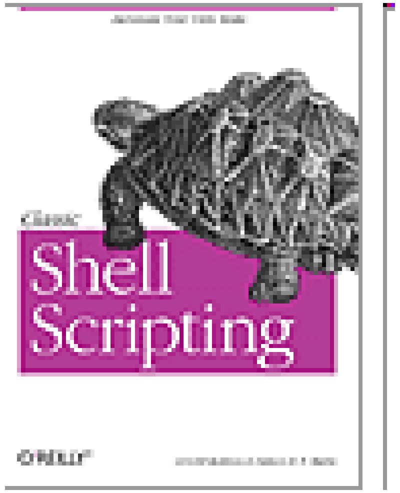
```
#!/bin/sh
 sum() {
   x=`expr $1 + $2`
   echo $x
   }

sum 5 3
echo "The sum of 4 and 7 is `sum 4 7`"
```

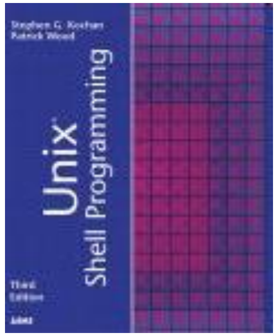
Hands-on Exercises

1. The simplest Hello World shell script – Echo command
2. Summation of two integers – If block
3. Summation of two real numbers – bc (basic calculator) command
4. Script to find out the biggest number in 3 numbers – If –elif block
5. Operation (summation, subtraction, multiplication and division) of two numbers – Switch
6. Script to reverse a given number – While block
7. A more complicated greeting shell script
8. Sort the given five numbers in ascending order (using array) – Do loop and array
9. Calculating average of given numbers on command line arguments – Do loop
10. Calculating factorial of a given number – While block
11. An application in research computing – Combining all above
12. Optional: Write own shell scripts for your own purposes if time permits

Reference Books



- **Class Shell Scripting**
  http://oreilly.com/catalog/9780596005955/
- **LINUX Shell Scripting With Bash**
  http://ebooks.ebookmall.com/title/linux-shell-scripting-with-bash-burtch-ebooks.htm
- **Shell Script in C Shell**
  http://www.grymoire.com/Unix/CshTop10.txt
- **Linux Shell Scripting Tutorial**
  http://www.freeos.com/guides/lsst/
- **Bash Shell Programming in Linux**
  http://www.arachnoid.com/linux/shell_programming.html
- **Advanced Bash-Scripting Guide**
  http://tldp.org/LDP/abs/html/
- **Unix Shell Programming**
  http://ebooks.ebookmall.com/title/unix-shell-programming-kochan-wood-ebooks.htm



Thanks

Q & A