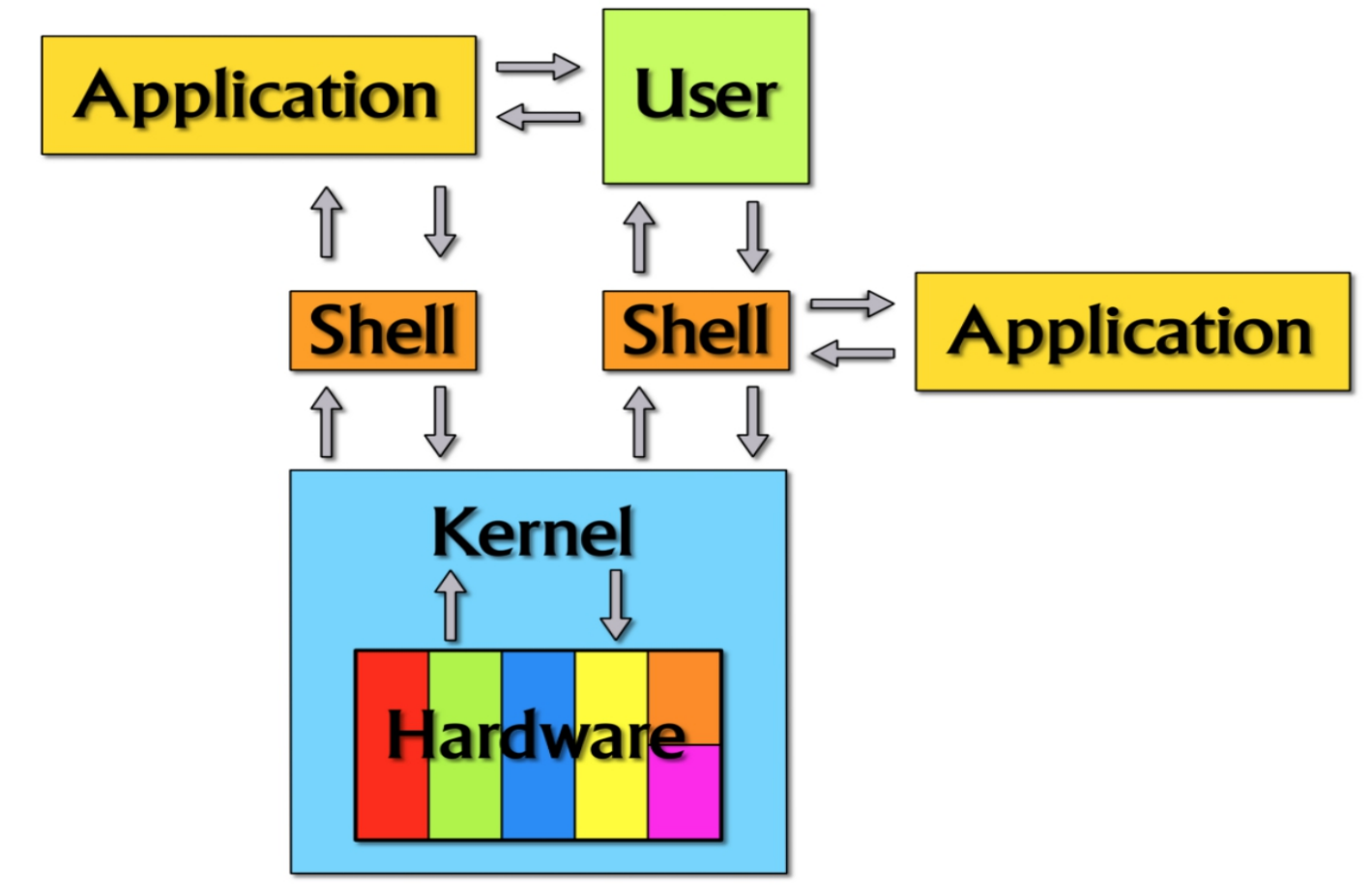
Lập trình Shell

Nguyễn Văn Hòa, Khoa KTCNMT
Đại học An Giang

Shell and Kernel



Nội dung

- Giới thiệu
  - UNIX/LINUX Shell
  - Basic Shell Scripting Structure
- Lập trình Shell
  - Biến
  - Phép toán
  - Cấu trúc điều khiển
- Các ví dụ minh họa

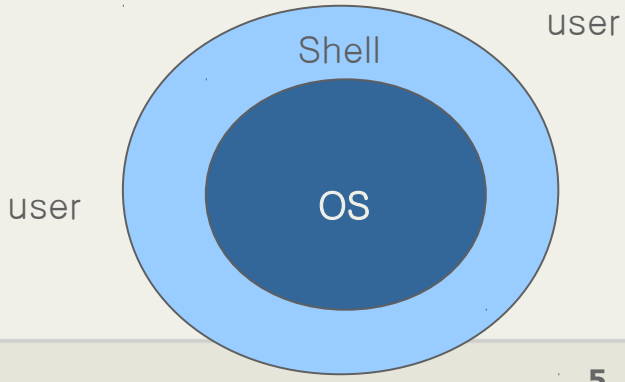
Tại sao phải lập trình Shell?

- Shell có thể dùng để xử lý số liệu, theo dõi chương trình, xử lý kết quả chương trình
- Tạo ra những lệnh riêng hữu ích
- Tiết kiệm thời gian xử lý công việc
- Thực hiện tự động các công việc
- Tự động hóa một phần trong quản trị hệ thống

“Shell” là gì?

- “Shell” là chương trình ở mức trên cùng của kernel nhằm cung cấp giao diện tương tác với OS
  - Là lệnh thông dịch
    - Built on top of the kernel
    - Enables users to run services provided by the UNIX OS
  - Có thể là chuỗi các lệnh trong một file (chương trình Shell) nhằm tiết kiệm thời gian gõ lại lệnh để thực hiện một công việc nào đó.
- Kiểm tra phiên bản Shell đang dùng

```
echo $SHELL
```



UNIX Shells

- sh Bourne Shell (Original Shell) (*Steven Bourne of AT&T*)
- bash Bourne Again Shell (*GNU Improved Bourne Shell*)
- csh C-Shell (C-like Syntax) (*Bill Joy of Univ. of California*)
- ksh Korn-Shell (Bourne+some C-shell)(*David Korn of AT&T*)
- tcsh Turbo C-Shell (More User Friendly C-Shell).
- To check shell:
  - $ `echo $SHELL` (shell is a pre-defined variable)

Nên dùng Shell nào?

- sh (Bourne shell) was considered better for programming
- csh (C-Shell) was considered better for interactive work.
- tcsh and korn were improvements on c-shell and bourne shell respectively.
- bash is largely compatible with sh and also has many of the nice features of the other shells
- On many systems such as our LINUX clusters sh is symbolically linked to bash, /bin/sh -> /bin/bash
- All Linux versions use the Bash shell (Bourne Again Shell) as the default shell
  - Bash/Bourn/ksh/sh prompt: $

Kịch bản Shell là gì?

- Kịch bản Shell (Shell script) là chuỗi các lệnh của Shell
- Kịch bản Shell bao gồm 2 phần
  - LINUX commands
  - Cú pháp của chương trình Shell

Viết kịch bản Shell

- Start vi | gedit scriptfilename.sh with the line

  #!/bin/sh
- All other lines starting with # are comments
  - make code readable by including comments
- Tell Unix that the script file is executable

  $ chmod u+x scriptfilename.sh

  $ chmod +x scriptfilename.sh
- Execute the shell-script

  $ ./scriptfilename.sh

Kịch bản Shell đơn giản

$ vi myfirstscript.sh

```
#! /bin/sh
# The first example of a shell script
directory=`pwd`
echo Hello World!
echo The date today is `date`
echo The current directory is $directory
```

$ chmod +x myfirstscript.sh

$ ./myfirstscript.sh

```
Hello World!
The date today is Mon Mar 8 15:20:09 EST 2010
The current directory is /netscr/shubin/test
```

Các kịch bản Shell

- Text files that contain sequences of UNIX commands , created by a text editor
- No compiler required to run a shell script, because the UNIX shell acts as an interpreter when reading script files
- After you create a shell script, you simply tell the OS that the file is a program that can be executed, by using the chmod command to change the files' mode to be executable
- Shell programs run less quickly than compiled programs, because the shell must interpret each UNIX command inside the executable script file before it is executed

Commenting

- Lines starting with # are comments except the very first line where #! indicates the location of the shell that will be run to execute the script.
- On any line characters following an unquoted # are considered to be comments and ignored.
- Comments are used to;
  - Identify who wrote it and when
  - Identify input variables
  - Make code easy to read
  - Explain complex code sections
  - Version control tracking
  - Record modifications

Dấu nháy

Dấu nháy dùng để chứa các khoảng trắng trong tham số. Có ba loại dấu nháy

“ : double quote: hiệu lực yếu, được dùng cho chuỗi có khoảng trắng

Ví dụ Myname = “Hương Nhi”

‘ : single quote: hiệu lực mạnh hơn, tên biến nằm trong dấu nháy này cũng sẽ bị vô hiệu hóa.

` : back quote. Lệnh bên trong dấu này sẽ được thực hiện trước và kết quả sẽ được thay thế ở stdout trước khi thực hiện lệnh khác

Example: `echo Today is:`date``

Lập trình Shell

- Các yếu tố cơ bản trong lập trình Shell
  - Shell variables: Các biến (trong đó chú ý các biến chuẩn)
  - Operators: Các phép toán số học
  - Logic structures: Biểu thức điều kiện, cấu trúc rẽ nhánh, cấu trúc lặp

Biến

- Sử dụng biến là cách đặt giá trị vào biến và truy xuất giá trị lưu trong biến
- Có ba loại biến khác nhau trong Shell
  - Global Variables: Environment and configuration variables, capitalized, such as HOME, PATH, SHELL, USERNAME, and PWD.
  - 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: Reversed for OS, shell programming, etc. such as positional parameters $0, $1 …

A few global (environment) variables

SHELL	Current shell
DISPLAY	Used by X-Windows system to identify the display
HOME	Fully qualified name of your login directory
PATH	Search path for commands
MANPATH	Search path for <man> pages
PS1 & PS2	Primary and Secondary prompt strings
USER	Your login name
TERM	terminal type
PWD	Current working directory

Referencing Variables

Variable contents are accessed using '$':

e.g. $ echo $HOME

$ echo $SHELL

To see a list of your environment variables:

```
$ printenv
```

or:

```
$ printenv | more
```

Khai báo Local Variables

- Variables can be defined and used in shell scripts.
- Variables in Shell Scripts are not typed (thông dịch).
- Examples :

```
a=1234  # a is NOT an integer, a string instead
b=$a+1 # will not perform arithmetic but be the string '1234+1'
b=`expr $a + 1`  will perform arithmetic so b is 1235 now.
```

Note : +,-,/,*,**, % operators are available.

```
b=abcde # b is string
b='abcde' # same as above but much safer.
b=abc  def # will not work unless 'quoted'
b='abc def' # i.e. this will work.
```

NOTE: không có khoảng trắng trước & sau "="

Variable List/Arrary

- To set a list element – square bracket
  var_array[index]=value | var_name=(values)
- To view a list element:
  echo $var_array[index]
- Example: myarray.sh
  #!/bin/sh
  Unix[0]= 'Debian'
  Unix[1]='Red hat'
  Unix[2]='Ubuntu'
  echo ${a[*]}
  echo ${a[0]}
  Chuyển sang mode thực thi: $ chmod +x myarray.sh
  $ ./myarray.sh (Results: ??)

Sử dụng tham số dòng lệnh

- Các tham số dòng lệnh được phân cách bởi ký tự trống
- Tên lệnh và các đối số được gán cho các biến $0, $1, …, $9.
- $0 This variable that contains the name of the script
- $1, $2, ….. $n 1st, 2nd 3rd 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)

Example:

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.

Biến

- vi myinputs.sh

```
#! /bin/sh
echo Total number of inputs: $#
echo First input: $1
echo Second input: $2
```

- chmod u+x myinputs.sh
- myinputs.sh HUSKER UNL CSE

```
Total number of inputs: 3
First input: HUSKER
Second input: UNL
```

Các phép toán

- expr supports the following operators:
  - arithmetic operators: +,-,*,/,%
  - comparison operators: <, <=, ==, !=, >=, >
  - boolean/logical operators: &, |
  - parentheses: (,)
  - precedence is the same as C, Java

Arithmetic Operators

- vi math.sh

```
#!/bin/sh
count=5
count=`expr $count + 1 `
echo $count
```

- chmod u+x math.sh
- ./math.sh

  6

Arithmetic operations in shell scripts

var++ ,var-- , ++var , --var	post/pre increment/decrement
+ , -	add subtract
* , / , %	multiply/divide, remainder
**	power of
! , ~	logical/bitwise negation
& , |	bitwise AND, OR
&& ||	logical AND, OR

Cấu trúc điều khiển

The four basic logic structures:

- 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

Conditional Statements (if constructs)

The most general form of the if construct is;

```
if command executes successfully
then
        execute command
elif this command executes successfully
then
        execute this command
        and execute this command
else
        execute default command
fi
```

However- elif and/or else clause can be omitted.

Examples

SIMPLE EXAMPLE:

```
if date | grep “Fri”
then
        echo “It’s Friday!”
fi
```

FULL EXAMPLE:

```
if   [  “$1”  ==  “Monday”  ]
then
        echo “The typed argument is Monday.”
elif [ “$1” == “Tuesday” ]
 then
        echo “Typed argument is Tuesday”
 else
        echo “Typed argument is neither Monday nor Tuesday”
fi
# Note: =  or == will both work in the test but == is better for readability.
```

Tests

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

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
```

File enquiry operations

-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.

Decision Logic

- **A simple example**

```
#!/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
```

Decision Logic

Another example:

```
#! /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:

```
#!/bin/bash
for i in 3 2 5 7
do
    echo " $i times 5 is  $((  $i  * 5 ))  "
done
```

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
  done
```

EXAMPLE:

```
#!/bin/bash
i=10
  while test "$i" -gt 0      # can also be  while  [ $i > 0 ]
  do
        echo $i
          i=`expr $i - 1`
  done
```

Looping Logic

- Example:

```
#!/bin/sh
for person in Bob Susan Joe Gerry
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
```

Switch/Case Logic

- The switch logic structure 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

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
```

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`"
```