Linux System Administration-I
CSE-4043

Chapter 2 : Scripting and the Shell

Nibedita Jagadev
Assistant Professor
Department of Computer Science and Engineering
SOA University, Bhubaneswar, Odisha, India
Email: nibeditajagadev@soa.ac.in

Contents

- Introduction
- Shell Basics
- Pipes and Redirection
- Unix Kernel
- The ps command displays active processes

Introduction

- A **Unix shell** is a command-line interpreter or **shell** that provides a traditional **Unix**-like command line user interface. Users direct the operation of the computer by entering commands as text for a command line interpreter to execute, or by creating text scripts of one or more such commands.
- A **shell script** is a computer program designed to be run by the Unix **shell**, a command-line interpreter. The various dialects of **shell scripts** are considered to be **scripting** languages. Typical operations performed by **shell scripts** include file manipulation, program execution, and printing text.
- Scripts standardize and automate the performance of administrative chores and free up admins' time for more important and more interesting tasks. In a sense, scripts are also a kind of low-rent documentation in that they act as an authoritative outline of the steps needed to complete a particular task.

Introduction

- In terms of complexity, administrative scripts run from simple ones that encapsulate a few static commands to major software projects that manage host configurations and administrative data for an entire site.
- Administrative scripts should emphasize programmer efficiency and code clarity rather than computational efficiency. This is not an excuse to be sloppy, but simply a recognition that it rarely matters whether a script runs in half a second or two seconds. Optimization can have an amazingly low return on investment, even for scripts that run regularly out of **cron.**
- For a long time, the standard language for administrative scripts was the one defined by the shell. Most systems' default shell is **bash (the "Bourne-again" shell),** but **sh (the original Bourne shell) and ksh (the Korn shell) are used on a few** UNIX systems.
- Shell scripts are typically used for light tasks such as automating a sequence of commands or assembling several filters to process data.

Introduction

- The shell is always available, so shell scripts are relatively portable and have few dependencies other than the commands they invoke.
- For more sophisticated scripts, it’s advisable to jump to a real programming language such as Perl or Python, both of which are well suited for administrative work. These languages incorporate a couple of decades’ worth of language design advancements relative to the shell, and their text processing facilities (invaluable to administrators) are very powerful.

Shell Basics

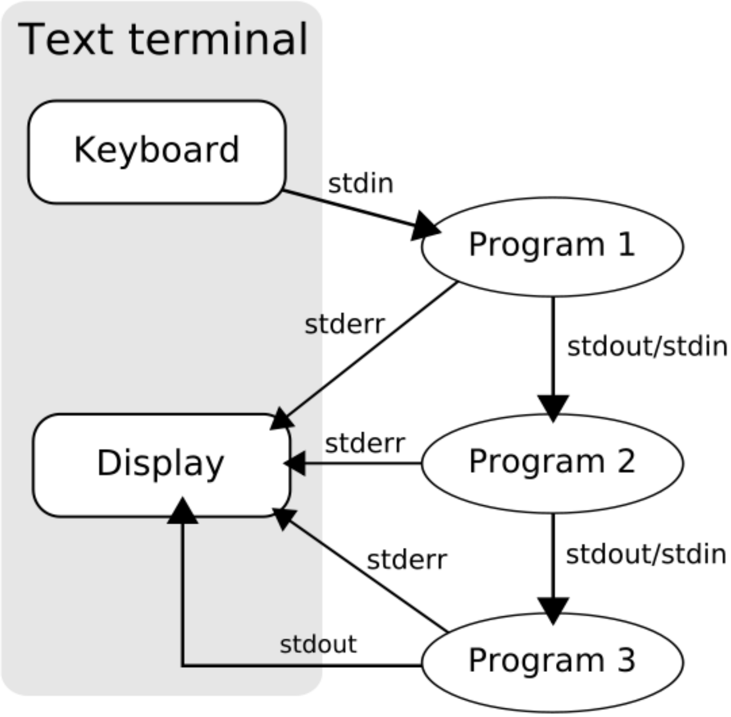
Command Editing

- <Control-E> goes to the end of the line
- <Control-A> to the beginning
- <Control-P> steps backward through recently executed commands and recalls them for editing.
- <Control-R> searches incrementally through your history to find old commands.
- For shell's command-line editing into **vi mode like this:**

  $ **set -o vi**
- and Type <Esc> to leave input mode and "i" to reenter it.
- For **emacs editing mode:** $ **set -o emacs**

Pipes and Redirection

- In **Unix/ Linux** computer operating systems, a pipeline is a sequence of processes chained together by their standard streams, so that the output of each process (stdout) feeds directly as input (stdin) to the next one.



Pipes and Redirection

- UNIX has a unified I/O model in which each channel is named with a small integer called a file descriptor. The exact number assigned to a channel is not usually significant, but STDIN, STDOUT, and STDERR are guaranteed to correspond to file descriptors 0, 1, and 2, so it’s safe to refer to these channels by number. In the context of an interactive terminal window, STDIN normally reads from the keyboard and both STDOUT and STDERR write their output to the screen.
- Most commands accept their input from STDIN and write their output to STDOUT. They write error messages to STDERR. This convention lets you string commands together like building blocks to create composite pipelines.

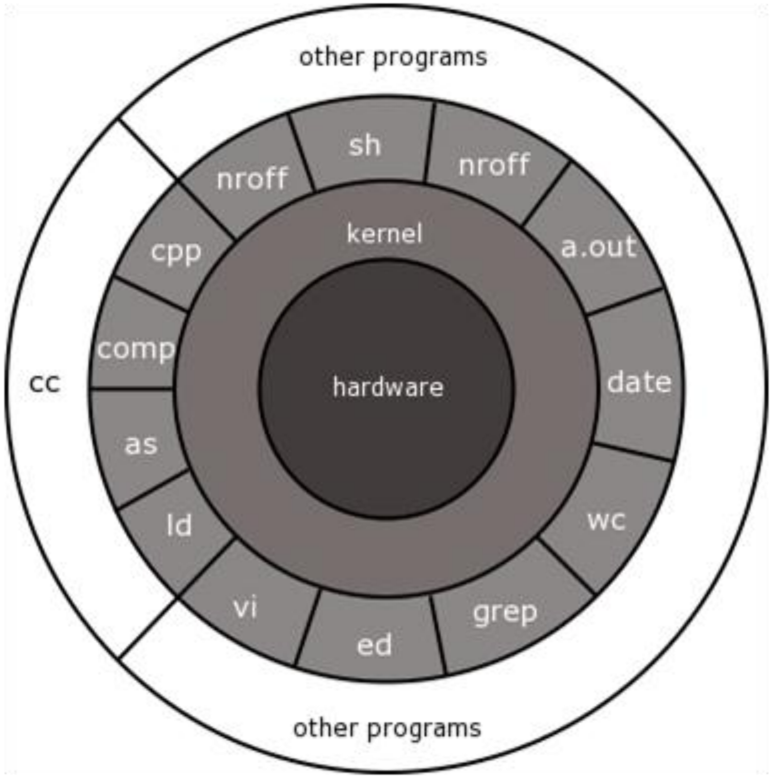
Pipes and Redirection

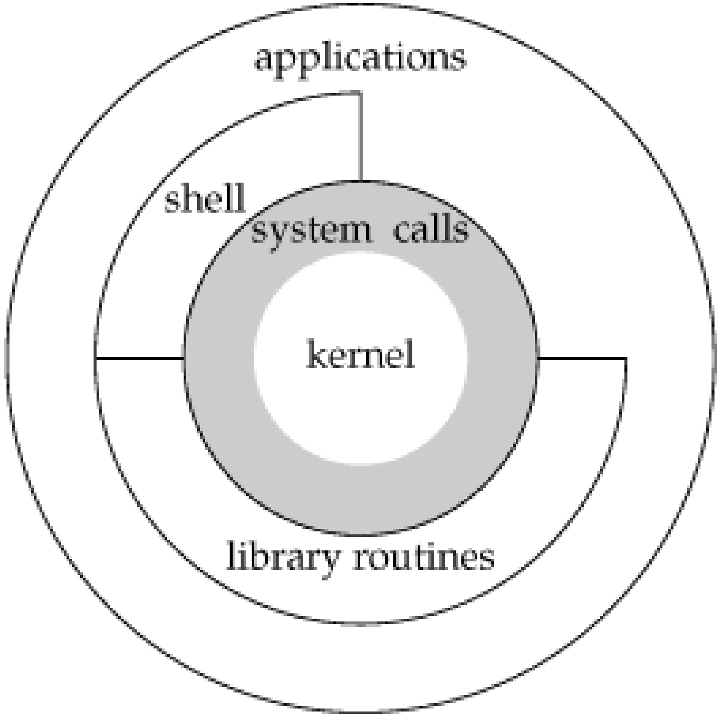
- Every process has at least three communication channels available to it: “standard input” (STDIN), “standard output” (STDOUT), and “standard error” (STDERR). The kernel sets up these channels on the process’s behalf, so the process itself doesn’t necessarily know where they lead. They might connect to a terminal window, a file, a network connection, or a channel belonging to another process, to name a few possibilities.

# NOTE: The **kernel** is the essential center of a computer operating system, the core that provides basic services for all other parts of the operating system. A synonym is nucleus. A **kernel** can be contrasted with a shell, the outermost part of an operating system that interacts with user commands.

Unix Kernel

- The **UNIX** operating system is a set of programs that act as a link between the computer and the user. The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the operating system or **kernel**.





Pipes and Redirection

- The shell interprets the symbols **<, >, and >> as instructions to reroute a command's** input or output to or from a file.
- A **< symbol connects the command's** STDIN to the contents of an existing file.
- The **> and >> symbols redirect STDOUT; > replaces the file's existing contents, and >> appends to them.**
- $ **echo "This is a test message." > /tmp/mymessage**
- stores a single line in the file **/tmp/mymessage, creating the file if necessary.**

- $ **mail -s "Mail test" johndoe < /tmp/mymessage**
- This command sends the emails the contents of that file to user johndoe.

Pipes and Redirection

- To redirect both STDOUT and STDERR to the same place, use the **>& symbol.**
- **To** redirect STDERR only, use **2>.**

#Note: The **find command illustrates why you might want to handle STDOUT and** STDERR separately because it tends to produce output on both channels, especially when run as an unprivileged user.

```
$ find / -name core
```

- usually results in so many "permission denied" error messages that genuine hits get lost in the clutter.

```
$ find / -name core 2> /dev/null
```

- To discard all the error messages

```
$ find / -name core > /tmp/corefiles 2> /dev/null
```

- This command line sends matching paths to **/tmp/corefiles, discards errors,** and sends nothing to the terminal window.

Pipes and Redirection

- To connect the STDOUT of one command to the STDIN of another, use the **|** symbol, commonly known as a pipe.

```
$ ps -ef | grep httpd
```

This runs **ps to generate a list of processes and pipes it through the grep command to select lines that contain the word httpd. Grep** searches the named input FILEs (or standard input if no files are named, or the file name - is given) for lines containing a match to the given PATTERN. By default, **grep** prints the matching lines. In addition, two variant programs egrep and fgrep are available. The output of grep is not redirected, so the matching lines come to the terminal window.

Pipes and Redirection

$ cut -d: -f7 < /etc/passwd | sort -u

- The cut command picks out the path to each user's shell from /etc/passwd. The list of shells is then sent through sort -u to produce a sorted list of unique values.

- **$ lpr /tmp/t2 && rm /tmp/t2**
- To execute a second command only if its precursor completes successfully, you can separate the commands with an **&& symbol. For example**
- removes **/tmp/t2 if and only if it is successfully queued for printing. Here, the** success of the **lpr command is defined as its yielding an exit code of zero, so the** use of a symbol that suggests "logical AND" for this purpose may be confusing if you're used to short-circuit evaluation in other programming languages.

Pipes and Redirection

- In a script, one can use a backslash to break a command onto multiple lines, helping to distinguish the error-handling code from the rest of the command pipeline:

```
cp --preserve --recursive /etc/* /spare/backup \
|| echo "Did NOT make backup“
```

- For the converse effect—multiple commands combined onto one line—you can use a semicolon as a statement separator.

The ps command displays active processes

Option	Description
-a	Displays all processes on a terminal, with the exception of group leaders.
-c	Displays scheduler data.
-d	Displays all processes with the exception of session leaders.
-e	Displays all processes.
-f	Displays a full listing.
-g*list*	Displays data for the *list* of group leader IDs.
-j	Displays the process group ID and session ID.
-l	Displays a long listing
-p*list*	Displays data for the *list* of process IDs.
-s*list*	Displays data for the *list* of session leader IDs.
-t*list*	Displays data for the *list* of terminals.
-u*list*	Displays data for the *list* of usernames.

Linux System Administration–I
CSE–4043

Scripting and the Shell

NIBEDITA JAGADEV

Department of CSE

Asst. Professor

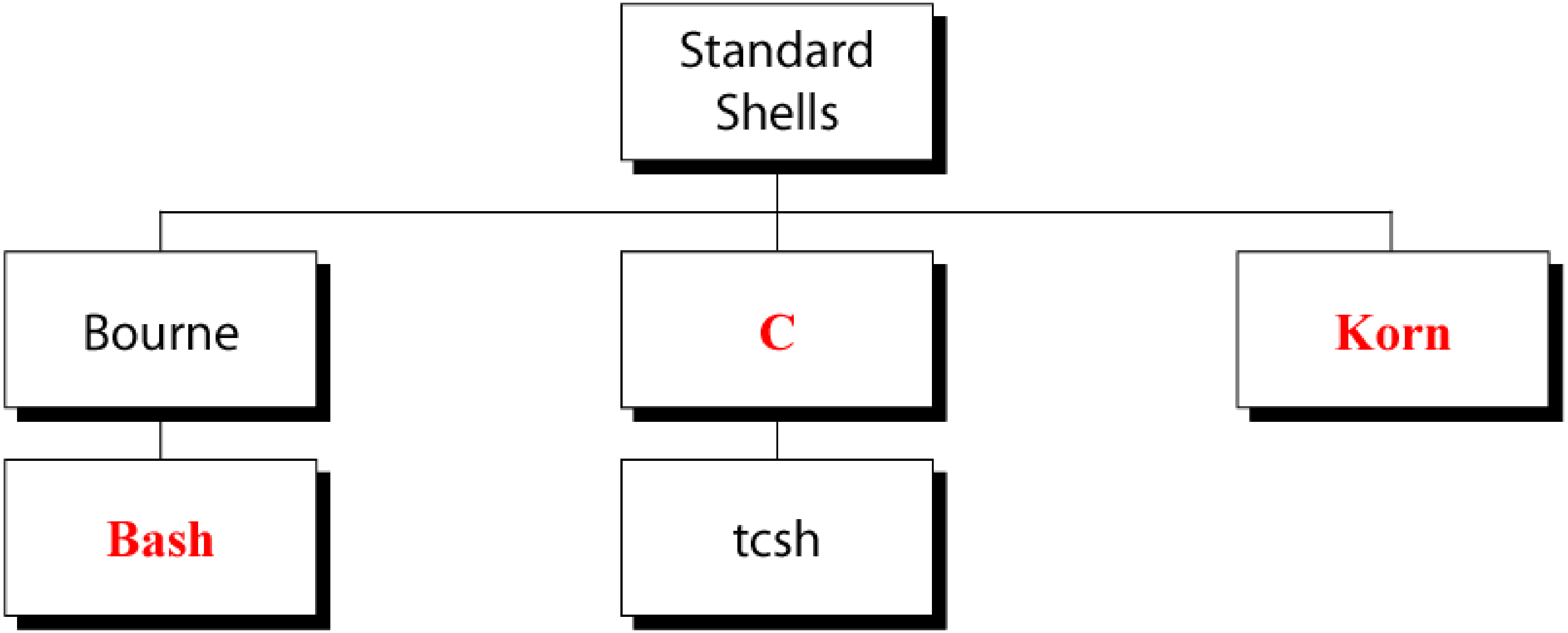
SOA Deemed to be University, Bhubaneswar, Odisha , India

nibeditajagadev@soa.ac.in

Contents

- Unix Command Interpreters
- Shell Program
- Shell Programming Library
- Steps To Create Shell Programs
- Shebang
- Use Of Bin Bash
- The /Bin Directory
- Programming Or Scripting
- Variables
- Warning
- Single And Double Quote
- The Export Command
- Environmental Variables
- Logname And Hostname

UNIX Command Interpreters



UNIX Command Interpreters

- **Bash** is a Unix **shell** and command language written by Brian Fox for the GNU Project as a free software replacement for the Bourne **shell**. First released in 1989, it has been distributed widely as the default **shell** for Linux distributions and Apple's mac OS (formerly OS X).
- 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).

Shell Program

- Shell programming is one of the most powerful features on any UNIX system
- If you cannot find an existing utility to accomplish a task, you can build one using a shell script
- A shell program contains high-level programming language features:
  - Variables for storing data
  - Decision-making control (e.g. if and case statements)
  - Looping abilities (e.g. for and while loops)
  - Function calls for modularity
- A shell program can also contain:
  - UNIX commands
  - Pattern editing utilities (e.g. grep, sed, awk)

Shell Programming Library

- Naming of shell programs and their output
  - Give a meaningful name
  - Program name example: find file . csh
  - Do not use: script1, script2
  - Do not use UNIX command names

- Repository for shell programs
  - If you develop numerous shell programs, place them in a directory (e.g. bin or shell progs)
  - Update your path to include the directory name where your shell programs are located

Steps to Create Shell Programs

- Specify shell to execute program
  - Script must begin with #! (pronounced "shebang") to identify shell to be executed

Examples:

```
#! /bin/sh                    (defaults to bash)
#! /bin/bash
#! /bin/csh
#! /usr/bin/tcsh
```

- Make the shell program executable
  - Use the "chmod" command to make the program/script file executable
- Formatting of shell programs
  - Indent areas (3 or 4 spaces) of programs to indicate that commands are part of a group
  - To break up long lines, place a \ at the end of one line and continue the comma nd on the next line
- Comments
  - Start comment lines with a pound sign (#)
  - Include comments to describe sections of your program
  - Help you understand your program when you look at it later

Steps of Programming

Guidelines:

use good names for

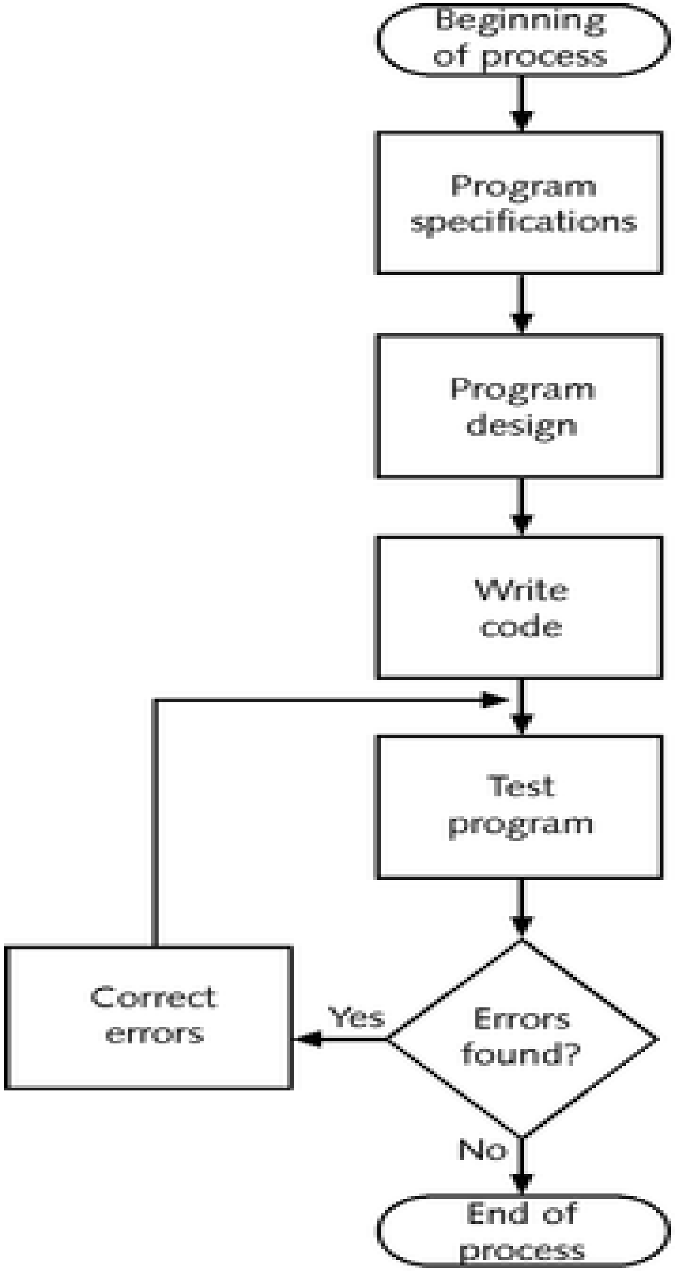
-script

-variables

use comments lines

-start with #

-use indentation to reflect logic and nesting



HELLO SCRIPT

- Example: “hello” Script

```
#! /bin/csh
echo "Hello $USER"
echo "This machine is `uname -n`"
echo "The calendar for this month is:"
cal
echo "You are running these processes:"
ps
```

SHEBANG

What is #! line in Linux?

- It is called a shebang or a "bang" line. It is nothing but the absolute path to the Bash interpreter. It consists of a number sign and an exclamation point character (#!), followed by the full path to the interpreter such as /bin/bash. All scripts under Linux execute using the interpreter specified on a first line. It is also called sha-bang hashbang, pound-bang, or hash-pling.

BIN BASH

- What is the use of Bin Bash?
- A script may specify #!/bin/bash on the first line, meaning that the script should always be run with bash, rather than another shell. /bin/sh is executable ,representing the system shell. Actually, it is usually implemented as a symbolic link pointing to the executable for whichever shell is the system shell.

The /bin Directory

- */bin* is a standard subdirectory of the *root directory* in Unix-like operating systems that contains the *executable* (i.e., ready to run) programs that must be available in order to attain minimal functionality for the purposes of *booting* (i.e., starting) and repairing a system.

- The root directory, which is designated by a forward slash (/), is the *top-level* directory in the hierarchy of directories (also referred to as the *directory tree*) on Unix-like operating systems. That is, it is the directory that contains all other directories and their subdirectories as well as all files on the system.

The /bin Directory

A directory in a Unix-like operating system is merely a special type of file that contains a list of the names of objects (i.e., files, links and directories) that appear to the user to be in it along with the corresponding inodes for each object. A file is a named collection of related information that appears to the user as a single, contiguous block of data and that is retained in storage (e.g., a hard disk drive or a floppy disk). An inode is a data structure on a filesystem that stores all the information about a filesystem object except its name and its actual data.

The /bin Directory

- A data structure is a way of storing data so that it can be used efficiently. A filesystem is the hierarchy of directories that is used to organize files on a computer system.
- The full names (also referred to as the absolute pathnames) of all of the subdirectories in the root directory begin with a forward slash, which shows their position in the filesystem hierarchy. In addition to /bin, some of the other standard subdirectories in the root directory include /boot, /dev, /etc, /home, /mnt, /usr, /proc and /var.
- Among the contents of /bin are the shells (e.g., bash and csh), ls, grep, tar, kill, echo, ps, cp, mv, rm, cat, gzip, ping, su and the vi text editor. These programs can be used by both the root user (i.e., the administrative user) and ordinary users.
- A list of all the programs in /bin can be viewed by using the ls command, which is commonly used to view the contents of directories, i.e.,
- ls /bin

The /bin Directory

- /bin is by default in PATH, which is the list of directories that the system searches for the corresponding program when a command is issued. This means that any *executable file* (i.e., runnable program) in /bin can be run just by entering the file name at the command line and then pressing the ENTER key. The contents of PATH can be seen by using the *echo* command as follows:
- echo $PATH

Programming or Scripting ?

- Bash is not only an excellent command line shell, but a scripting language in itself. Shell scripting allows us to use the shell's abilities and to automate a lot of tasks that would otherwise require a lot of commands.
- 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 file. This executable is not easily ported into different operating systems.
- A scripting language also starts from source code, but is not compiled into an executable file. 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.

The First Bash Program

- There are two major text editors in Linux:
  - vi, emacs (or xemacs).
- So fire up a text editor; for example:

```
 $ vi &
```

and type the following inside it:

```
#!/bin/bash
echo “Hello World”
```

- The first line tells Linux to use the bash interpreter to run this script. We call it hello.sh. Then, make the script executable:

```
$ chmod 700 hello.sh
$ ./hello.sh
Hello World
```

Variables

- We can use variables as in any programming languages. Their values are always stored as strings, but there are mathematical operators in the shell language that will convert variables to numbers for calculations.
- We have no need to declare a variable, just assigning a value to its reference will create it.
- Example
- #!/bin/bash
- STR=“Hello World!”
- echo $STR
- Line 2 creates a variable called STR and assigns the string "Hello World!" to it. Then the value of this variable is retrieved by putting the '$' in at the beginning.

Warning !

- The shell programming language does not type-cast its variables. This means that a variable can hold number data or character data.

- count=0
- count=Sunday

- Switching the TYPE of a variable can lead to confusion for the writer of the script or someone trying to modify it, so it is recommended to use a variable for only a single TYPE of data in a script.

- \ is the bash escape character and it preserves the literal value of the next character that follows.

- $ ls *
- ls: *: No such file or directory

Single and Double Quote

- When assigning character data containing spaces or special characters, the data must be enclosed in either single or double quotes.
- Using double quotes to show a string of characters will allow any variables in the quotes to be resolved
- $ var=“test string”
- $ newvar=“Value of var is $var”
- $ echo $newvar
- Value of var is test string
- Using single quotes to show a string of characters will not allow variable resolution
- $ var=’test string’
- $ newvar=’Value of var is $var’
- $ echo $newvar
- Value of var is $var

The export command

- The export command puts a variable into the environment so it will be accessible to child processes. For instance:

- If the child modifies x, it will not modify the parent’s original value.

Linux System Administration–I
CSE-4043

Scripting and the Shell

NIBEDITA JAGADEV

Department of CSE

Asst. Professor

SOA Deemed to be University, Bhubaneswar, Odisha , India

nibeditajagadev@soa.ac.in

Contents

- Environmental Variables
- Logname
- Hostname
- Arithmetic Evaluation
- Conditional Statements
- Expressions
- Shell Parameters
- Case Statement
- Iteration Statements
- Using Arrays with Loops
- A C-like for loop
- Debugging on the entire script

Environmental Variables

- There are two types of variables:
- Local variables
- Environmental variables
- Environmental variables are set by the system and can usually be found by using the env command. Environmental variables hold special values. For instance:

```
            $ echo $SHELL
            /bin/bash
            $ echo $PATH
/usr/X11R6/bin:/usr/local/bin:/bin:/usr/bin
```

- Environmental variables are defined in /etc/profile, /etc/profile.d/ and ~/.bash_profile. These files are the initialization files and they are read when bash shell is invoked.
- When a login shell exits, bash reads ~/.bash_logout
- The startup is more complex; for example, if bash is used interactively, then /etc/bashrc or ~/.bashrc are read. See the man page for more details.

Environmental Variables

- LOGNAME: contains the user name
- HOSTNAME: contains the computer name.

- PS1: sequence of characters shown before the prompt

\t	hour
\d	date
\w	current directory
\W	last part of the current directory
\u	user name
\$	prompt character

Logname

In computer software, logname (stands for Login Name) is a program in Unix and Unix-like operating systems that prints the name of the user executing the command. It corresponds to the LOGNAME variable in the system-state environment. The logname system call and command appeared for the first time in UNIX System III.

Hostname

- **Hostname** is used to display the system's DNS name, and to display or set its **hostname** or NIS (Network Information Services) domain name. When called without any arguments, **hostname** will display the name of the system as returned by the get hostname function.

Arithmetic Evaluation

- The let statement can be used to do mathematical functions:

```
$ let X=10+2*7
$ echo $X
24
$ let Y=X+2*4
$ echo $Y
32
```

- An arithmetic expression can be evaluated by $[expression] or $((expression))

```
$ echo “$((123+20))”
143
$ VALORE=$[123+20]
$ echo “$[123*$VALORE]”
17589
```

Arithmetic Evaluation

- Available operators: +, -, /, *, %
- Example

```
$ cat arithmetic.sh
#!/bin/bash
echo -n “Enter the first number: ”; read x
echo -n “Enter the second number: ”; read y
add=$(($x + $y))
sub=$(($x - $y))
mul=$(($x * $y))
div=$(($x / $y))
mod=$(($x % $y))
# print out the answers:
echo “Sum: $add”
echo “Difference: $sub”
echo “Product: $mul”
echo “Quotient: $div”
echo “Remainder: $mod”
```

Conditional Statements

Conditionals let us decide whether to perform an action or not, this decision is taken by evaluating an expression. The most basic form is:

```
if [ expression ];
then
        statements
elif [ expression ];
then
        statements
else
        statements
fi
```

the elif (else if) and else sections are optional

Put spaces after [and before], and around the operators and operands.

Expressions

- An expression can be: String comparison, Numeric comparison, File operators an Expressions
- Logical operators are represented by [expression]:
- String Comparisons:

= compare if two strings are equal

!= compare if two strings are not equal

-n evaluate if string length is greater than zero

-z evaluate if string length is equal to zero

- Examples:

[s1 = s2] (true if s1 same as s2, else false)

[s1 != s2] (true if s1 not same as s2, else false)

[s1] (true if s1 is not empty, else false)

[-n s1] (true if s1 has a length greater then 0, else false)

[-z s2] (true if s2 has a length of 0, otherwise false

Expressions

- Number Comparisons:

-eq compare if two numbers are equal

-ge compare if one number is greater than or equal to a number

-le compare if one number is less than or equal to a number

-ne compare if two numbers are not equal

-gt compare if one number is greater than another number

-lt compare if one number is less than another number

- Examples:

[n1 -eq n2] (true if n1 same as n2, else false)

[n1 -ge n2] (true if n1greater then or equal to n2, else false)

[n1 -le n2] (true if n1 less then or equal to n2, else false)

[n1 -ne n2] (true if n1 is not same as n2, else false)

[n1 -gt n2] (true if n1 greater then n2, else false)

[n1 -lt n2] (true if n1 less then n2, else false

Expressions

- **Files operators:**

-d check if path given is a directory

-f check if path given is a file

-e check if file name exists

-r check if read permission is set for file or directory

-s check if a file has a length greater than 0

-w check if write permission is set for a file or directory

-x check if execute permission is set for a file or directory

- **Examples:**

[-d fname] (true if fname is a directory, otherwise false)

[-f fname] (true if fname is a file, otherwise false)

[-e fname] (true if fname exists, otherwise false)

[-s fname] (true if fname length is greater then 0, else false)

[-r fname] (true if fname has the read permission, else false)

[-w fname] (true if fname has the write permission, else false)

[-x fname] (true if fname has the execute permission, else false)

Expressions

- Logical operators:

! negate (NOT) a logical expression

-a logically AND two logical expressions

-o logically OR two logical expressions

Example:

```
#!/bin/bash
   echo -n “Enter a number 1 < x < 10:”
   read num
   if [ “$num” -gt 1 –a “$num” -lt 10 ];
   then
         echo “$num*$num=$(($num*$num))”
   else
         echo “Wrong insertion !”
   fi
```

Expressions

- Logical operators:

&&   logically AND two logical expressions

| | logically OR two logical expressions

Example

```
#!/bin/bash
    echo -n "Enter a number 1 < x < 10: "
    read num
    if [ “$number” -gt 1 ] && [ “$number” -lt 10 ];
    then
        echo “$num*$num=$(($num*$num))”
    else
        echo “Wrong insertion !”
    fi
```

Shell Parameters

- Positional parameters are assigned from the shell’s argument when it is invoked. Positional parameter “N” may be referenced as “${N}”, or as “$N” when “N” consists of a single digit.
- Special parameters

$# is the number of parameters passed

$0 returns the name of the shell script running as well as its location in the file system

$* gives a single word containing all the parameters passed to the script

$@ gives an array of words containing all the parameters passed to the script

```
$ cat sparameters.sh
#!/bin/bash
echo “$#; $0; $1; $2; $*; $@”
$ sparameters.sh arg1 arg2
2; ./sparameters.sh; arg1; arg2; arg1 arg2; arg1 arg2
```

Case Statement

- Used to execute statements based on specific values. Often used in place of an if statement if there are a large number of conditions.
- Value used can be an expression
- each set of statements must be ended by a pair of semicolons;
- a *) is used to accept any value not matched with list of values

```
case $var in
    val1)
        statements;;
    val2)
        statements;;
    *)
        statements;;
    esac
```

#Note: The **esac** keyword is indeed a required delimiter to end a case statement in bash and most shells used on **Unix**/Linux excluding the csh family. The original Bourne shell was created by Steve Bourne who previously worked on ALGOL68. This language invented this reversed word technique to delimit blocks.

Iteration Statements

- The for structure is used when you are looping through a range of variables.

```
for var in list
   do
    statements
   done
```

- statements are executed with var set to each value in the list.
- Example

```
#!/bin/bash
   let sum=0
   for num in 1 2 3 4 5
     do
       let “sum = $sum + $num”
     done
   echo $sum
```

Using Arrays with Loops

- n the bash shell, we may use arrays. The simplest way to create one is using one of the two subscripts:

```
pet[0]=dog
pet[1]=cat
pet[2]=fish
pet=(dog cat fish)
```

- We may have up to 1024 elements. To extract a value, type ${arrayname[i]}

```
$ echo ${pet[0]}
    dog
```

- To extract all the elements, use an asterisk as:

```
echo ${arrayname[*]}
```

- We can combine arrays with loops using a for loop:

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

A C-like for loop

- An alternative form of the for structure is

```
for (( EXPR1 ; EXPR2 ; EXPR3 ))
   do
        statements
   done
```

- First, the arithmetic expression EXPR1 is evaluated. EXPR2 is then evaluated repeatedly until it evaluates to 0. Each time EXPR2 evaluates to a non-zero value, statements are executed and EXPR3 is evaluated.

Debugging on the entire script

- When things don't go according to plan, you need to determine what exactly causes the script to fail. Bash provides extensive debugging features. The most common is to start up the subshell with the -x option, which will run the entire script in debug mode. Traces of each command plus its arguments are printed to standard output after the commands have been expanded but before they are executed.
- Note that the added comments are not visible in the output of the script.
- Bash provides two options which will give useful information for debugging

-x : displays each line of the script with variable substitution and before execution

-v : displays each line of the script as typed before execution

Linux System Administration–I
CSE-4043

Scripting and the Shell

NIBEDITA JAGADEV

Department of CSE

Asst. Professor

SOA Deemed to be University, Bhubaneswar, Odisha , India

nibeditajagadev@soa.ac.in

Contents

- While Statements
- Continue Statements
- Break Statements
- Until Statements
- Functions
- Example (case.sh)
- Manipulating Strings
- Parameter Substitution

While Statements

- The while structure is a looping structure. Used to execute a set of commands while a specified condition is true. The loop terminates as soon as the condition becomes false. If condition never becomes false, loop will never exit.

```
while expression
    do
          statements
    done
```

```
$ cat while.sh
    #!/bin/bash
    echo –n “Enter a number: ”; read x
    let sum=0; let i=1
    while [ $i –le $x ]; do
     let “sum = $sum + $i”
           i=$i+1
    done
  echo “the sum of the first $x numbers is: $sum”
```

Continue Statements

- The continue command causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.
- Break Statements
- The break command terminates the loop (breaks out of it).

Continue Statements

```
$ cat continue.sh
#!/bin/bash
    LIMIT=19
    echo
    echo “Printing Numbers 1 through 20 (but not 3 and 11)”
    a=0
    while [ $a -le “$LIMIT” ]; do
     a=$(($a+1))
     if [ “$a” -eq 3 ] | | [ “$a” -eq 11 ]
     then
            continue
     fi
     echo -n “$a ”
    done
```

Break Statements

```
$ cat break.sh
#!/bin/bash
   LIMIT=19
   echo
   echo “Printing Numbers 1 through 20, but something happens after 2 … ”
   a=0
   while [ $a -le “$LIMIT” ]
   do
    a=$(($a+1))
    if [ “$a” -gt 2 ]
    then
   break
    fi
  echo -n “$a ”
 done
 echo; echo; echo
 exit 0
```

Until Statements

- The until structure is very similar to the while structure. The until structure loops until the condition is true. So basically it is “until this condition is true, do this”.

```
until [expression]
   do
        statements
   done
```

Until Statements

```
$ cat countdown.sh
    #!/bin/bash
    echo “Enter a number: ”; read x
    echo ; echo Count Down
    until [ “$x” -le 0 ]; do
    echo $x
    x=$(($x –1))
    sleep 1
  done
echo ; echo GO !
```

Functions

- Functions make scripts easier to maintain. Basically it breaks up the program into smaller pieces. A function performs an action defined by you, and it can return a value if you wish.

```
#!/bin/bash
    hello()
    {
    echo “You are in function hello()”
    }
    echo “Calling function hello()…”
    hello
    echo “You are now out of function hello()”
```

- In the above, we called the hello() function by name by using the line: hello . When this line is executed, bash searches the script for the line hello(). It finds it right at the top, and executes its contents.

Functions

```
$ cat function.sh
#!/bin/bash
function check() {
if [ -e "/home/$1" ]
then
  return 0
else
  return 1
fi
}
echo “Enter the name of the file: ” ; read x
if check $x
then
  echo “$x exists !”
else
  echo “$x does not exists !”
fi.
```

Example (case.sh)

```
$ cat case.sh
#!/bin/bash
    echo -n “Enter a number 1 < x < 10: ”
  read x
    case $x in
        1) echo “Value of x is 1.”;;
        2) echo “Value of x is 2.”;;
        3) echo “Value of x is 3.”;;
        4) echo “Value of x is 4.”;;
        5) echo “Value of x is 5.”;;
        6) echo “Value of x is 6.”;;
        7) echo “Value of x is 7.”;;
        8) echo “Value of x is 8.”;;
        9) echo “Value of x is 9.”;;
        0 | 10) echo “wrong number.”;;
        *) echo “Unrecognized value.”;;
    esac
```

Manipulating Strings

- Bash supports a number of string manipulation operations.

${#string} gives the string length

${string : position} extracts sub-string from $string at $position

${string: position: length} extracts $length characters of sub-string from $string at $position

- Example

```
$ st=0123456789
$ echo ${#st}
    10
$ echo ${st:6}
    6789
$ echo ${st:6:2}
    67
```

Parameter Substitution

- Manipulating and/or expanding variables

${parameter-default}, if parameter not set, use default.

```
$ echo ${username-`whoami`}
    alice
$ username=bob
$ echo ${username-`whoami`}
    bob
```

${parameter=default}, if parameter not set, set it to default.

```
$ unset username
$ echo ${username=`whoami`}
$ echo $username
    alice
```

${parameter+value}, if parameter set, use value, else use null string.

```
$ echo ${username+bob}
    bob
```

Thank You