Introduction to Linux Basics

Part-II Shell Scripting

Georgia Advanced Computing Resource Center
University of Georgia
Zhuofei Hou, HPC Trainer
zhuofei@uga.edu
Outline

• What is GACRC?
• What are Linux Shell and Shell Scripting?
• Shell Scripting Syntax Basics
• Real Shell Scripting Examples
What is GACRC?

Who Are We?
- Georgia Advanced Computing Resource Center
- Collaboration between the Office of Vice President for Research (OVPR) and the Office of the Vice President for Information Technology (OVPIT)
- Guided by a faculty advisory committee (GACRC-AC)

Why Are We Here?
- To provide computing hardware and network infrastructure in support of high-performance computing (HPC) at UGA

Where Are We?
- [http://gacrc.uga.edu](http://gacrc.uga.edu) (Web)
- [http://wiki.gacrc.uga.edu](http://wiki.gacrc.uga.edu) (Wiki)
- [http://gacrc.uga.edu/help/](http://gacrc.uga.edu/help/) (Web Help)
- [https://wiki.gacrc.uga.edu/wiki/Getting_Help](https://wiki.gacrc.uga.edu/wiki/Getting_Help) (Wiki Help)
What are Linux Shell and Shell Scripting?

Linux: A full-fledged operating system with 4 major parts:

I. Kernel: Low-level OS, handling files, disks, RAM, networking, etc.

II. Supplied Programs: Web browsing, Audio, Video, DVD burning......

III. Shell: A command-line user interface for a user to type and run commands:
   - Bourne Shell (sh)
   - Korn Shell (ksh)  UNIX standard shells
   - C Shell (csh)
   - Bourne-Again Shell (bash)  → Linux default shell

IV. X: A graphical system providing graphical user interface(GUI)
What are Linux Shell and Shell Scripting?

- **Linux Shell**: A place to type and run commands on Linux
  - Command-line interface for user to type commands
  - Command interpreter to run commands
  - Programming environment for scripting

- **Linux default**: Bourne-Again Shell (bash)

- **To open a shell on**:

<table>
<thead>
<tr>
<th>Local Linux/Mac</th>
<th>Terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Windows</td>
<td>SSH Secure Client or Cygwin</td>
</tr>
<tr>
<td>Remote Linux machine</td>
<td>A shell will run immediately when log in</td>
</tr>
</tbody>
</table>
What are Linux Shell and Shell Scripting?

- **Linux Shell Script**: A *text file running as a program*, to accomplish tasks on Linux that a single command cannot
  - Shell Variables *(assignment, exporting)*
  - Expansion (*~, $, ``, $(( ))*)
  - Quoting *(`,`*, `""*, `" '"*)
  - Commands *(|, ;)*
  - Redirection *(>, >>, 2>, 2>&1, >&, < )*
  - Flow Control *(if-then-else)*
  - Loops *(for, while)*

- **Linux Shell Scripting**: *Programming with Linux shell scripts*
Shell Scripting Syntax Basics – Shell Variables

- **Variable Assignment:** name=value (NO space! name = value is wrong!)

```bash
$ var1=kiwi       # all values held in variables are strings! var1="kiwi"
$ echo $var1      # echo prints the value of var1 to screen
$ kiwi

$ var2=7         # same as var2="7"
$ echo $var2
$ 7

$ var3=$var1+7   # same as var3="kiwi+7"
$ echo $var3
$ kiwi+7

$ var4=10        # same as var4="10"
$ echo $var2+$var4
$ 7+10
```
Shell Scripting Syntax Basics – Shell Variables

- Variable Exporting: `export var` (var is a global environment variable for use in shell’s child processes running in subshells; Otherwise, it is a local variable!)

```bash
$ var1=kiwi
$ export var2=apple # var2=apple; export var2
$ printenv var1    # printenv prints env variables
$
$ printenv var2
$ apple
```

- Numeric Expression Evaluation: `expr` or `$(...)`

```bash
$ var1=10
$ var2=20
$ expr $var1 + $var2    # space and $ are required!
$ 30
$ echo $((var1+var2))   # space and $ are not required!
$ 30
```
Shell Scripting Syntax Basics – Shell Variables

- bash **automatically** sets some *special* shell variables at startup time (Note: Some of them may be *environment variables*)

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOME*</td>
<td>Home directory of the current user</td>
</tr>
<tr>
<td>PATH*</td>
<td>Search path for commands (colon-separated dirs in which shell looks for commands)</td>
</tr>
<tr>
<td>PWD*</td>
<td>Current working directory</td>
</tr>
<tr>
<td>SHELL*</td>
<td>Default shell currently being used</td>
</tr>
<tr>
<td>USER*</td>
<td>Current user’s name</td>
</tr>
<tr>
<td>UID</td>
<td>Numeric user ID of the current user</td>
</tr>
<tr>
<td>LD_LIBRARY_PATH*</td>
<td>Shared library search path</td>
</tr>
</tbody>
</table>
Shell Scripting Syntax Basics – Shell Variables

- Why we have those automatically set *shell variables*?
  - Configure your working environment on Linux as you wish!

- Example: `.bash_profile` for interactive login shell

```bash
if [ -f ~/.bashrc ]; then
  . ~/.bashrc
fi

export PATH=$PATH:$HOME/bin

# User specific environment and startup programs
export PATH=$PATH:$HOME/bin

# Zhuofei 2015-05-29
export PATH=$PATH:$HOME/scripts
```
Shell Scripting Syntax Basics – Shell Variables

- Suggestion 1: "$var" to prevent runtime errors in script

```bash
$ var="My Document"    # "My Document" is a subdirectory
$ cd $var             # same as cd My Document, 2 args
$ -bash: cd: My: No such file or directory
$ cd "$var"           # same as cd "My Document", 1 args
My Document$
```

- Suggestion 2: ${var} to prevent unexpected behavior

```bash
$ var="apple"
$ echo "Mary has 3 $vars"    # variable vars is empty!
$ Mary has 3
$ echo "Mary has 3 ${var}s"  # ${var} is not working!
$ Mary has 3 ${apple}s
$ echo "Mary has 3 ${var}s"  # ${var} is working!
$ Mary has 3 apples
```
Shell Scripting Syntax Basics – Expansion

- **Tilde Expansion (Home Expansion): ~**
  ```
  $ cd ~username  # home directory associated username
  $ cd ~          # replaced by $HOME
  $ cd ~/         # same as above
  ```

- **Variable Expansion: $**
  ```
  $ var=24
  $ echo ${var}th  # outputs 24th; ${var} to prevent unexpected behavior!
  ```

- **Command Substitution: `command` (` is back quota!)**
  ```
  $ cd `pwd`       # same as cd /home/abclab/jsmith/workingDir
  ```

- **Numeric Expansion: $((expression))**
  ```
  $ echo $(( (5+3*2)-1)/2 ))  # outputs 5; space is not required!
  $ var1=24 ; var2=10          # ; for a sequence of commands
  $ echo $((var1+var2))        # outputs 34
  ```
Shell Scripting Syntax Basics – Quoting

- Linux special characters:
  \` ~ ! # % ^ & * ( ) - + / \ | ; ' " , . < > ? { 

- Quoting rules in bash:

  1. All special characters are disabled by enclosing double quotes “”, except for !, $, `, \, and {
  2. All special characters are disabled by enclosing single quotes ‘’
  3. All special characters are **forcedly** disabled by a preceding backslash \

Shell Scripting Syntax Basics – Quoting

➢ Quoting Examples

```bash
$ FRUIT=apples
$ echo 'I like $FRUIT'          # $ is disabled by ''
$ I like $FRUIT
$ echo "I like $FRUIT"         # $ is not disabled by " "
$ I like apples
$ echo "I like \$FRUIT"        # $ is disabled forcedly by preceding \n$ I like $FRUIT
$ echo `pwd`                   # ` is disabled by ` ``
$ `pwd`
$ echo `pwd`                   # ` is not disabled by " "
$ /home/abclab/jsmith
```
Shell Scripting Syntax Basics – Commands

- Pipeline `command1 | command2 | ...` connects std output of `command1` to the std input of `command2`, and so on (Demonstration)

```bash
$ ls -l | more
$ ls -l | grep ".sh"
$ ps aux | awk '{if($1=="zhuofei") print $0}' | more
$ qstat -u "*" | awk '{print $4}' | sort | less
$ qstat -u "*" | grep 'qw' | awk 'BEGIN{n=0} {n++} END{printf "%d jobs waiting on queue\n", n}'
```

- List `command1 ; command2 ; ... ;` simply runs commands in sequence on a single command line (Demonstration)

```bash
$ pwd ; ls
$ cd .. ; ls
$ mkdir ./subdir ; cd ./subdir ; touch file1 ; ls
```
Shell Scripting Syntax Basics – Redirection

- **Standard output redirection:** `>` and `>>`

  ```
  $ ls > outfile  # std output of a command is written to outfile
  $ ls >> outfile  # std output of a command is appended to outfile
  $ ./myprog > outfile  # std output of a program is written to outfile
  ```

- **Standard error redirection:** `2>`, `2>&1` and `>>&`

  ```
  $ ./myprog > outfile 2> errorfile  # std output and error ➔ separate files
  $ ./myprog > outfile 2>&1  # std output and error ➔ single file
  $ ./myprog >& outfile  # same as above
  ```

- **Standard input redirection:** `<`

  ```
  $ ./myprog < infile  # std input is from infile
  ```

- **General usage:**

  ```
  $ ./myprog < infile > outfile 2>&1
  ```
Shell Scripting Syntax Basics – Flow Control

- If-fi Block:

```bash
if [ test expression ] : if test expression is evaluated to be true
then
    body1
else
    body2
fi
```

- Example (Demonstration):

```bash
echo "Please enter you name:"
read name # read a line from standard input
if [ "$name" == "zhuofei" ] # true if strings are equal
then
    echo "Hello, ${name}!"
else
    echo "Hi, ${name}, you are not zhuofei!"
fi
```
# Shell Scripting Syntax Basics – Flow Control

<table>
<thead>
<tr>
<th>Test Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e file</td>
<td>True if file exists</td>
</tr>
<tr>
<td>-d or -f file</td>
<td>True if file exists and is a directory or a regular file</td>
</tr>
<tr>
<td>-r or -w or -x file</td>
<td>True if file exists and is readable or writable or executable</td>
</tr>
<tr>
<td>-s file</td>
<td>True if file exists and has a nonzero size</td>
</tr>
<tr>
<td>file1 -nt or -ot file2</td>
<td>True if file1 is newer or older than file2</td>
</tr>
<tr>
<td>-z or -n string</td>
<td>True if the length of string is zero or nonzero</td>
</tr>
<tr>
<td>str1 == str2</td>
<td>True if the strings are equal</td>
</tr>
<tr>
<td>str1 != str2</td>
<td>True if the strings are not equal</td>
</tr>
<tr>
<td>arg1 OP arg2</td>
<td>OP is one of –eq, -ne, -lt, -le, -gt, or -ge. Arg1 and arg2 may be +/- integers</td>
</tr>
<tr>
<td>! expr</td>
<td>True if expr is false</td>
</tr>
<tr>
<td>expr1 -a expr2</td>
<td>True if both expr1 AND expr2 are true</td>
</tr>
<tr>
<td>expr1 -o expr2</td>
<td>True if either expr1 OR expr2 is true</td>
</tr>
</tbody>
</table>
Shell Scripting Syntax Basics – Loops

- **for Loop:**
  ```bash
  for variable in list
do
    body
done
  ```

- **while Loop:**
  ```bash
  while [ test expression ]
do
    body
done
  ```

- **Example (Demonstration):**
  ```bash
  for file in *.doc *.docx
do
    echo "$file is a MS word file!"
done

  i=1
  while [ $i -le 10 ]
do
    echo $i
    i=`expr $i + 1`
done
  ```
#!/bin/bash
SUBDIR=`pwd`
CTR=1
for sub in ${SUBDIR}/*.* ; do
  if [ "`basename ${sub}`" != "`basename $0`" ] ; then
    qsub -q rcc-30d ${sub} > ${SUBDIR}/outfile_${CTR}
    echo "`basename ${sub}` submitted!"
    CTR=$(($CTR+1))
  fi
done
printf "\nTotally %d jobs submitted!\n\n" $((CTR-1))
qstat -u `id -un`
Real Shell Scripting Examples

- Example 2: a serial job submission script on zcluster

```
#!/bin/bash
cd `pwd`
time ./myprog < myin > myout
```

- Example 3: a MPI job submission script on zcluster *(default MPICH2 and PGI compilers)*

```
#!/bin/bash
cd `pwd`
export LD_LIBRARY_PATH=/usr/local/mpich2/1.4.1p1/pgi123/lib:${LD_LIBRARY_PATH}
mpirun -np $NSLOTS ./myprog
```

https://wiki.gacrc.uga.edu/wiki/Running_Jobs_on_zcluster
Thank You!