LSU INFORMATION TECHNOLOGY SERVICES

LONI

Basic Shell Scripting

Wei Feinstein

HPC User Services

LSU HPC & LON

sys-help@loni.org

February 2018

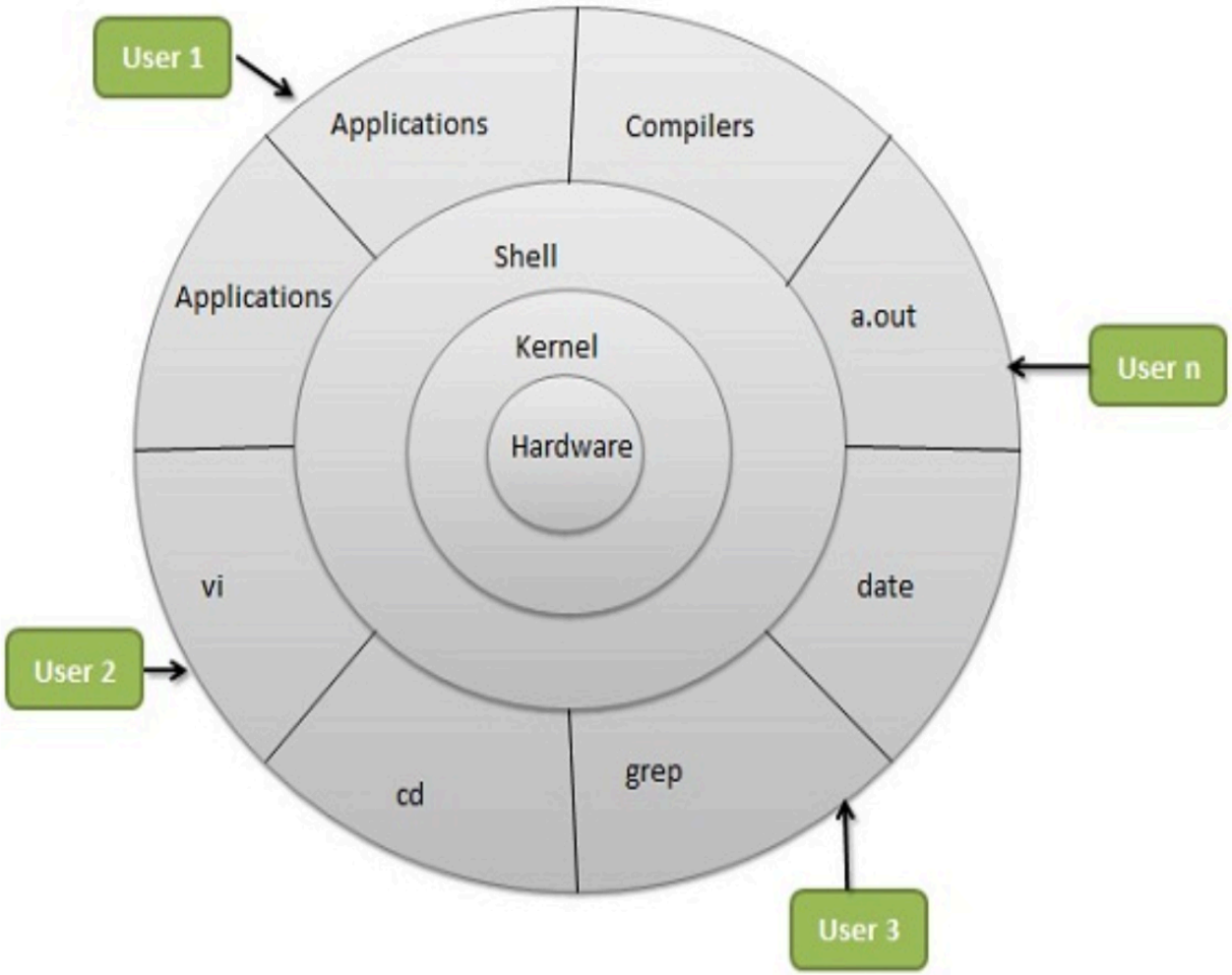
XSEDE

Extreme Science and Engineering Discovery Environment

Outline

- Introduction to Linux Shell
- Shell Scripting Basics
  - Variables
  - Quotations
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

Linux System Architecture



Linux Shell

What is a Shell

- An application running on top of the kernel and provides a powerful interface to the system
- Process user's commands, gather input from user and execute programs
- Types of shell with varied features
  - sh
  - csh
  - ksh
  - bash
  - tcsh

Shell Comparison

Software	sh	csh	ksh	bash	tcsh
Programming language	y	y	y	y	y
Shell variables	y	y	y	y	y
Command alias	n	y	y	y	y
Command history	n	y	y	y	y
Filename autocompletion	n	y*	y*	y	y
Command line editing	n	n	y*	y	y
Job control	n	y	y	y	y

*: not by default

http://www.cis.rit.edu/class/simg211/unixintro/Shell.html

What can you do with a shell?

- Check the current shell
  - `echo $SHELL`
- List available shells on the system
  - `cat /etc/shells`
- Change to another shell
  - `exec sh`
- Date and time
  - `date`
- wget: get online files
  - wget https://ftp.gnu.org/gnu/gcc/gcc-7.1.0/gcc-7.1.0.tar.gz
- Compile and run applications
  - `gcc hello.c –o hello`
  - `./hello`

Outline

- Introduction to Linux Shell
- Shell Scripting Basics
  - Variables
  - Quotations
- Beyond Basic Shell Scripting
- Advanced Text Processing Commands (grep, sed, awk)

Shell Scripting

- Script: a program written for a software environment to automate execution of tasks
  - A series of shell commands put together in a file
  - When the script is executed, those commands will be executed one line at a time automatically

- The majority of script programs are “quick and dirty”, where the main goal is to get the program written quickly
  - May not be as efficient as programs written in C and Fortran

Script Example (~/.bashrc)

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
export PATH=$HOME/packages/eFindsite/bin:$PATH
export LD_LIBRARY_PATH=$HOME/packages/eFindsite/lib:$LD_LIBRARY_PATH
alias qsubI="qsub -I -X -l nodes=1:ppn=20 -l walltime=01:00:00 –A
my_allocation“
alias lh="ls -altrh"
```

Hello World

```
#!/bin/bash
# A script example
echo “Hello World”
```

1. #!: "Shebang” line to instruct which interpreter to use. In the current example, bash. For tcsh, it would be: #!/bin/tcsh
2. All comments begin with "#".
3. Print "Hello World!" to the screen.

Variables

- Variable names
  - Must start with a letter or underscore
  - Number can be used anywhere else
  - Do not use special characters such as @,#,%,$
  - Case sensitive
  - Allowed: VARIABLE, VAR1234able, var_name, _VAR
  - Not allowed: 1var, %name, $myvar, var@NAME
- Two types of variables:
  - Global variables (ENVIRONMENT variables)
  - Local variables (user defined variables)

Global Variables

- Environment variables provide a simple way to share configuration settings between multiple applications and processes in Linux
  - Using all uppercase letters
  - Example: PATH, LD_LIBRARY_PATH, DISPLAY etc.
- To reference a variable, prepend $ to the name of the variable
- Example: $PATH, $LD_LIBRARY_PATH, $DISPLAY etc.
- **`printenv/env`** list the current environmental variables in your system.

List of Some Environment Variables

PATH	A list of directory paths which will be searched when a command is issued
LD_LIBRARY_PATH	colon-separated set of directories where libraries should be searched for first
HOME	indicate where a user's home directory is located in the file system.
PWD	contains path to current working directory.
OLDPWD	contains path to previous working directory.
TERM	specifies the type of computer terminal or terminal emulator being used
SHELL	contains name of the running, interactive shell.
PS1	default command prompt
PS2	Secondary command prompt
HOSTNAME	The systems host name
USER	Current logged in user's name
DISPLAY	Network name of the X11 display to connect to, if available.

Editing Variables

- Assign values to variables

Type	sh/ksh/bash	csh/tcsh
Shell (local)	`name=value`	`set name=value`
Environment (global)	`export name=value`	`setenv name value`

- Shell variables is only valid within the current shell, while environment variables are valid for all subsequently opened shells.
- Example: useful when running a script, where exported variables (global) at the terminal can be inherited within the script.

With export	Without export

$ export v1=one
$ bash
$ echo $v1
→one
	$ v1=one
	$ bash
	$ echo $v1
	→

Quotations

- Single quotation
  - Enclosed string is read literally
- Double quotation
  - Enclosed string is expanded
- Back quotation
  - Enclose string is executed as a command

Quotation - Examples

```
[wfeinste@mike1    str1='echo $USER'
[wfeinste@mike1    echo $str1
 -> echo $USER
[wfeinste@mike1    str2="echo $USER"
[wfeinste@mike1    echo $str2
->echo wfeinste
[wfeinste@mike1    str3=`echo $USER`
[wfeinste@mike1    echo $str3
  -> wfeinste
```

Special Characters (1)

`#`	Start a comment line.
`$`	Indicate the name of a variable.
`\`	Escape character to display next character literally
`{ }`	Enclose name of variable
`;`	Command separator. Permits putting two or more commands on the same line.
`;;`	Terminator in a case option
`.`	"dot" command, equivalent to `source` (for bash only)

Special Characters (2)

`$?`	Exit status variable.
`$$`	Process ID variable.
`[]`	Test expression, eg. if condition
`[[ ]]`	Test expression, more flexible than [ ]
`$[], $(())`	Integer expansion
`||, &&, !`	Logical OR, AND and NOT

Outline

- Introduction to Linux Shell
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

Integer Arithmetic Operations

Operation	Operator
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Exponentiation	`** (bash only)`
Modulo	`%`

Integer Arithmetic Operations

- $((…)) or $[…] commands
  - Addition: x = $((1+2))
  - Multiplication: echo $[$x*$x]
- let command: let c=$x + $x
- expr command: expr 10 / 2 (space required)
- C-style increment operators:
  - let c+=1 or let c--

Floating-Point Arithmetic Operations

GNU basic calculator (bc) external calculator

- Add two numbers

```
echo "3.8 + 4.2" | bc
```

- Divide two numbers and print result with a precision of 5 digits:

```
echo "scale=5; 2/5" | bc
```

- Convert btw decimal and binary numbers

```
echo “ibase=10; obase=2; 10” |bc
```

- Call bc directly:

```
bc <<< “scale=5; 2/5”
```

Outline

- Introduction to Linux Shell
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Command Line Arguments
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

Arrays Operations (1)

- Initialization

```
declare -a my_array
my_array=("Alice" "Bill" "Cox" "David")
my_array[0]="Alice";
my_array[1]="Bill"
```

- Bash supports one-dimensional arrays
  - Index starts at 0
  - No space around "="
- Reference an element

```
${my_array[i]}
```

- Print the whole array

```
${my_array[@]}
```

- Length of array

```
${#my_array[@]}
```

Array Operations (2)

- Add an element to an existing array
  - `my_array=(first ${my_array[@]})`
  - `my_array=("${my_array[@]}" last)`
  - `my_array[4]=(“Nason”)`
- Copy an array name to an array user
  - `new_array=(${my_array[@]})`
- Concatenate two arrays
  - `two_arrays=(${my_array[@]} ${new_array[@]})`

Array Operations (3)

- Delete the entire array
  - `unset my_array`
- Delete an element to an existing array
  - `unset my_array[0]`

Outline

- Introduction to Linux Shell
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

Flow Control

- Shell scripting languages execute commands in sequence similar to programming languages such as C and Fortran
  - Control constructs can change the order of command execution
- Control constructs in bash
  - Conditionals: if-then-else
  - Loops: for, while, until
  - Switches: case

if statement

- if/then construct test whether the exit status of a list of commands is 0, and if so, execute one or more commands

```
if [ condition ]; then
   Do something
elif [ condition 2 ] ; then
   Do something
else
   Do something
fi
```

  - Strict spaces between condition and the brackets (bash)

File Operations

Operation	bash
File exists	`if [ -e test ]`
File is a regular file	`if [ -f test]`
File is a directory	`if [ -d /home ]`
File is not zero size	`if [ -s test ]`
File has read permission	`if [ -r test ]`
File has write permission	`if [ -w test ]`
File has execute permission	`if [ -x test ]`

Integer Comparisons

Operation	bash
Equal to	`if [ 1 -eq 2 ]`
Not equal to	`if [ $a -ne $b ]`
Greater than	`if [ $a -gt $b ]`
Greater than or equal to	`if [ 1 -ge $b ]`
Less than	`if [ $a -lt 2 ]`
Less than or equal to	`if [ $a -le $b ]`

String Comparisons

Operation	bash
Equal to	`if [ $a == $b ]`
Not equal to	`if [ $a != $b ]`
Zero length or null	`if [ -z $a ]`
Non zero length	`if [ -n $a ]`

Logical Operators

Operation	Example
! (NOT)	`if [ ! -e test ]`

&& (AND)	`if [ -f test] && [ -s test ]`
	`if [[ -f test && -s test ]]`
	`if ( -e test && ! -z test )`

| (OR)	`if [ -f test1 ] || [ -f test2 ]` `if [[ -f test1 || -f test2 ]]`

if condition examples

Example 1:

```
read input
if [ $input == "hello" ]; then
   echo hello;
else echo wrong ;
fi
```

Example 2

```
touch test.txt
if [ -e test.txt ]; then
   echo “file exist”
elif [ ! -s test.txt ]; then
   echo “file empty”;
fi
```

What happens after

```
echo “hello world” >> test.txt
```

Loop Constructs

- A loop is a block of code that iterates a list of commands as long as the loop control condition stays true
- Loop constructs

  `for`, `while` and `until`

for loop examples

Exmaple1:

```
for arg in `seq 1 4`
do
   echo $arg;
   touch test.$arg
done
```

How to delete test files using a loop?

```
rm test.[1-4]
```

Example 2:

```
for file in `ls /home/$USER`
do
   cat $file
done
```

While Loop

- The `while` construct test for a condition at the top of a loop and keeps going as long as that condition is true.
- In contrast to a `for` loop, a `while` is used when loop repetitions is not known beforehand.

```
read counter
while [ $counter -ge 0 ]
do let counter--
   echo $counter
done
```

Until Loop

- The `until` construct test a condition at the top of a loop, and stops looping when the condition is met (opposite of `while` loop)

```
read counter
until [ $counter -lt 0 ]
do let counter--
   echo $counter
done
```

Switching Constructs - bash

- The `case` constructs are technically not loops since they do not iterate the execution of a code block

```
#!/bin/sh
echo "Please talk to me ..."
while :
do
  read INPUT_STRING
  case $INPUT_STRING in
    hello)
        echo "Hello yourself!"
        ;;
    bye)
        echo "See you again!"
        break
        ;;
    *)
        echo "Sorry, I don't understand"
        ;;
  esac
Done
echo "That's all folks!"
```

Outline

- Introduction to Linux Shell
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

Functions

- A function is a code block that implements a set of operations. Code reuse by passing parameters
- By default all variables are global.
- Modifying a variable in a function changes it in the whole script.
- Create a local variables using the local command, which is invisible outside the function

```
local var=value
local varName
```

Functions example

```
#!/bin/bash
fun1(){
        local x_local=10
        x_global=100
}
x_global=10
echo "\nglobal initial x_global = $x_global"
fun1
echo "\nlocal x_local = $x_local"
echo "\nglobal final x_global = $x_global\n”
```

`$sh fun1.sh`

```
        global initial x_global = 10
        local x_local =
        global final x_global = 100
```

Pass Arguments to Bash Scripts

- All parameters can be passed at runtime and accessed via `$1, $2, $3…`
- $0: the shell script name
- `$* or $@:` all parameters passed to a function
- `$#:` number of positional parameters passed to the function
- $?: exist code of last cmd
- $$: PID of current process
- Array variable called `FUNCNAME` contains the names of all shell functions currently in the execution call stack.

Parameter example

```
#!/bin/bash
a=$1
b=$2
fun_mul(){
   fun_mul=$(($a*$b))
   echo ${FUNCNAME[0]}
}
echo "There are $# params $1 $2 passed in"
fun_mul
echo "\nProduct of $1 and $2 is $fun_mul\n"
echo "exit code=$? processID=$$  param=$*"
```

```
$ sh fun_param.sh 3 5
    There are 2 params 3 5 passed in
    fun_mul
    Product of 3 and 5 is 15
    exit code=0 processID=21459  param=3 5
```

Outline

- Introduction to Linux Shell
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

Advanced Text Processing Commands

- `grep`
- `sed`
- `awk`

grep & egrep

- **`grep:`** Unix utility that searches through either information piped to it or files.
- **`egrep:`** extended grep, same as `grep -E`
- **`zgrep:`** compressed files.

- Usage: `grep <options> <search pattern> <files>`
- Options:

`-i`	ignore case during search
`-r,-R`	search recursively
`-v`	invert match i.e. match everything except *`pattern`*
`-l`	list files that match *`pattern`*
`-L`	list files that do not match *`pattern`*
`-n`	prefix each line of output with the line number within its input file.
`-A num`	print `num` lines of trailing context after matching lines.
`-B num`	print `num` lines of leading context before matching lines.

grep Examples

- Search files containing the word `bash` in current directory

```
grep bash *
```

- Search files NOT containing the word `bash` in current directory

```
grep -v bash *
```

- Repeat above search using a case insensitive pattern match and print line number that matches the search pattern

```
grep -in bash *
```

- Search files not matching certain name pattern

```
ls | grep –vi fun
```

grep Examples

```
100  Thomas  Manager    Sales        $5,000
200  Jason     Developer  Technology  $5,500
300  Raj       Sysadmin   Technology  $7,000
500  Randy     Manager    Sales        $6,000
```

- grep OR

```
grep 'Man\|Sales' employee.txt
-> 100  Thomas  Manager    Sales      $5,000
   300  Raj     Sysadmin   Technology $7,000
   500  Randy   Manager    Sales      $6,000
```

- grep AND

```
grep –i 'sys.*Tech' employee.txt
–> 100300  Raj    Sysadmin   Technology  $7,000
```

sed

- "stream editor" to parse and transform information
  - information piped to it or from files
- line-oriented, operate one line at a time and allow regular expression matching and substitution.
- `s` substitution command

sed commands and flags

Flags	Operation	Command	Operation
-e	combine multiple commands	s	substitution
-f	read commands from file	g	global replacement
-h	print help info	p	print
-n	disable print	i	ignore case
-V	print version info	d	delete
-r	use extended regex	G	add newline
		w	write to file
		x	exchange pattern with hold buffer
		h	copy pattern to hold buffer
		;	separate commands

sed Examples

```
#!/bin/bash

# My First Script

echo "Hello World!”
```

sed Examples (1)

- Add flag -e to carry out multiple matches.

```
cat hello.sh | sed -e 's/bash/tcsh/g' -e 's/First/Second/g'
#!/bin/tcsh
# My Second Script
echo "Hello World!"
```

- Alternate form

```
sed 's/bash/tcsh/g; s/First/Second/g' hello.sh

#!/bin/tcsh
# My Second Script
echo "Hello World!"
```

- The default delimiter is slash (/), can be changed

```
sed 's:/bin/bash:/bin/tcsh:g' hello.sh

#!/bin/tcsh
# My First Script
echo "Hello World!"
```

sed Examples (2)

- Delete blank lines from a file

```
sed '/^$/d' hello.sh

#!/bin/bash
# My First Script
echo "Hello World!"
```

- Delete line `n` through `m` in a file

```
sed '2,4d' hello.sh

#!/bin/bash
echo "Hello World!"
```

sed Examples (3)

- Insert a blank line below every line matches *pattern*

```
sed '/First/G' hello.sh

#!/bin/bash

# My First Script


echo "Hello World!"
```

- Insert a blank line above and below every line matches *pattern*

```
sed '/First/{x;p;x;G}' hello.sh

#!/bin/bash


# My First Script


echo "Hello World!"
```

sed Examples (4)

- Replace-in-place with a backup file

```
sed –i.bak '/First/Second/i' hello.sh
```

- echo with sed

```
$ echo "shell scripting" | sed "s/[si]/?/g"
$ ?hell ?cr?pt?ng

$ echo "shell scripting 101" | sed "s/[0-9]/#/g"
$  shell scripting ###
```

awk

- The `awk` text-processing language is useful for tasks such as:
  - Tallying information from text files and creating reports from the results.
  - Adding additional functions to text editors like "vi".
  - Translating files from one format to another.
  - Creating small databases.
  - Performing mathematical operations on files of numeric data.
- `awk` has two faces:
  - It is a utility for performing simple text-processing tasks, and
  - It is a programming language for performing complex text-processing tasks.

How Does awk Work

- `awk` reads the file being processed line by line.
- The entire content of each line is split into columns with space or tab as the delimiter.
- `$0` Print the entire line, use.
- `NR` #records (lines)
- `NF` #fields or columns in the current line.
- By default the field delimiter is space or tab. To change the field delimiter use the `-F<delimiter>` command.

awk Syntax

`awk pattern {action}`

`pattern` decides when `action` is performed

`Actions:`

- Most common action: `print`
- Print file dosum.sh:

```
awk '{print $0}' dosum.sh
```

- Print line matching files in all `.sh` files in current directory:

```
awk '/bash/{print $0}' *.sh
```

```
uptime
11:18am up 14 days 0:40, 5 users, load average: 0.15, 0.11,
0.17

uptime | awk '{print $0}'
11:18am up 14 days 0:40, 5 users, load average: 0.15, 0.11,
0.17

uptime | awk '{print $1,NF}'
11:18am 12

uptime | awk '{print NR}'
1

uptime | awk -F, '{print $1}'
11:18am up 14 days 0:40

for i in $(seq 1 3); do touch file${i}.dat ; done
for i in file* ; do
> prefix=$(echo $i | awk -F. '{print $1}')
> suffix=$(echo $i | awk -F. '{print $NF}')
> echo $prefix $suffix $i; done

file1 dat file1.dat
file2 dat file2.dat
file3 dat file3.dat
```

Awk Examples

- Print list of files that are bash script files

```
awk '/^#\!\/bin\/bash/{print $0, FILENAME}' *
→   #!/bin/bash Fun1.sh
    #!/bin/bash fun_pam.sh
    #!/bin/bash hello.sh
    #!/bin/bash parm.sh
```

- Print extra lines below patterns

```
awk '/sh/{print;getline;print}' <hello.sh
#!/bin/bash
```

Getting Help

- User Guides
  - LSU HPC: http://www.hpc.lsu.edu/docs/guides.php#hpc
  - LONI: http://www.hpc.lsu.edu/docs/guides.php#loni
- Documentation: http://www.hpc.lsu.edu/docs
- Archived tutorials: http://www.hpc.lsu.edu/training/archive/tutorials.php
- Contact us
  - Email ticket system: sys-help@loni.org
  - Telephone Help Desk: 225-578-0900

Upcoming trainings

March 07, 2018: Hands-On Practice Session
March 14, 2018: Introduction to R
March 21, 2018: Parallel Computing with Matlab
April 04, 2018: Data Visualization in R
April 11, 2018: Introduction to Python
April 18, 2018: Deep Learning Software