



Linux Shell Basics

IT 4423

Unix/Linux Administration

J.G. Zheng
Spring 2012



Overview

◆ Scripting mode

- Command line scripting
- Script files

◆ Scripting basics

- Input/output and piping
- Variables and data types

Shell

- ◆ A Unix/Linux shell is a piece of software (command-line interpreter) that provides an interface for users to access the services of the OS kernel

- ◆ Major shells
 - Bourne shell compatibles
 - ◆ Bourne shell: /bin/sh
 - ◆ Bash (Bourne-Again shell): /bin/bash
 - C Shell (csh)
 - See the /etc/shells file for valid shells (not installed shells)

- ◆ Ubuntu (and most Linux distros) uses bash as the default shell
 - You can switch to a different shell at any time use the "chsh" command

Scripting

◆ Scripting vs. Programming

◆ Scripting language in Linux

- Shell (Bash)
- Perl
- Python

◆ Scripting mode

- Command line scripting
- Script files

Command Line Scripting

- ◆ Write scripts directly in the shell at the command prompt
- ◆ Command editing
 - Use up arrow key to get previous commands
 - Ctrl + A to go to the beginning of the command
 - Ctrl + E to go to the end of the command
- ◆ Multi line commands
 - Use “\” to indicate a soft return and continue on the next line
- ◆ Multiple commands on one line
 - Use “;” to separate commands

Shell Built-in Commands

- ◆ Shell built-ins are commands interpreted by the shell directly (no separate executable files)
 - cd
 - pwd
 - type
 - echo
 - alias

- ◆ Use "type" to see the command type

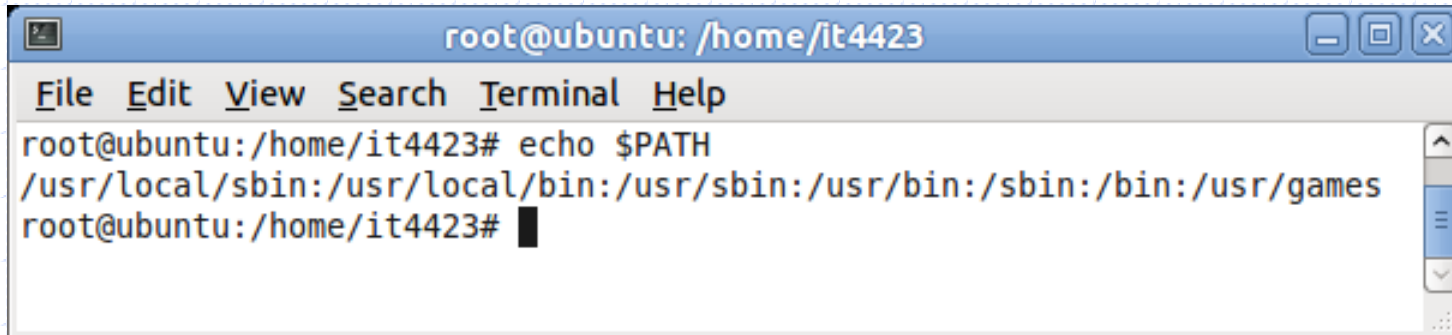
Command Alias

- ◆ An alias is a (usually short) name that the shell translates into another (usually longer) name or (complex) command.
 - Aliases allow you to define new commands.
- ◆ Example
 - `alias ls='ls -l'`
 - `alias cp='cp -i'`
 - `alias dir='ls -l'`
 - Enter just "alias" to check the current aliases defined
- ◆ When an alias hides the original command
 - Use "" (quotation mark) around the original command to avoid alias
- ◆ Remove alias
 - Use the `unalias` command followed by the alias name

Command Search Path

◆ Search path

- These are the directories to search when a command is entered without path.
- Paths are stored in the \$PATH environment variable

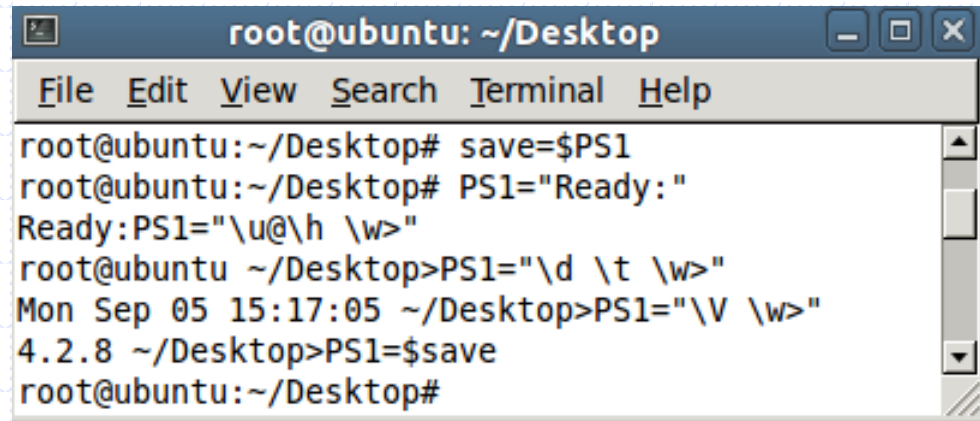


```
root@ubuntu: /home/it4423
File Edit View Search Terminal Help
root@ubuntu:/home/it4423# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
root@ubuntu:/home/it4423#
```

Changing Command Prompt

- ◆ Command prompt is usually used to display useful environment information such as
 - Current user, directory, date, etc.

- ◆ Change the "PS1" variable to change command prompt

A terminal window titled 'root@ubuntu: ~/Desktop' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following sequence of commands and outputs:

```
root@ubuntu:~/Desktop# save=$PS1
root@ubuntu:~/Desktop# PS1="Ready:"
Ready:PS1="\u@\h \w>"
root@ubuntu ~/Desktop>PS1="\d \t \w>"
Mon Sep 05 15:17:05 ~/Desktop>PS1="\V \w>"
4.2.8 ~/Desktop>PS1=$save
root@ubuntu:~/Desktop#
```

- ◆ Reference

- <http://www.linuxselfhelp.com/howtos/Bash-Prompt/Bash-Prompt-HOWTO-2.html>

Other Environmental Variables

◆ Prompt statement

- \$PS1 - command prompt
- \$PS2 - command continuation prompt

◆ Other commonly used ones

- \$SHELL - current shell
- \$HOME - home directory
- \$PWD - current working directory
- \$PATH - command search path

◆ Use "printenv" command to show all environmental variables

Input and Output

- ◆ Every process has at least 3 communication channels available
 - "standard input" (STDIN)
 - "standard output" (STDOUT)
 - "standard error" (STDERR)
- ◆ These channels are setup by the kernel, so the process itself doesn't necessarily know them
 - Most commands accept their input from STDIN and write their output to STDOUT. They write error messages to STDERR
- ◆ In the context of an interactive terminal window
 - STDIN normally reads from the keyboard
 - STDOUT and STDERR write their output to the screen

Reroute from/to Files

- ◆ Use ">" to redirect screen output to files
 - Use ">>" to append to a file rather than overwrite

```
#>echo "hello, world" > file1  
#>echo "hello, world" >> file1
```

- ◆ Use "<" to get input from a file

```
#>read variable1 < file1
```

- ◆ Use "2>" to redirect errors to a file
 - Use "/dev/null" if errors should be ignored

```
#>badcommand 2> file1  
#>badcommand 2> /dev/null
```

Pipe

◆ Pipe operator: |

- To connect the STDOUT of one command to the STDIN of another

```
#>ls -lat | head -2
```

◆ Filter

- Any well-behaved command that reads STDIN and writes STDOUT can be used as a filter (that is, a component of a pipeline) to process data.
- Common filter commands:
 - ◆ cut, sort, grep, wc, head, tail, tee

More Pipe Examples

◆ ls | more

◆ ls | wc -l

◆ ls -l | grep d

I/O Command: echo

- ◆ Use echo command to send results to "STDOUT"

```
#>echo "hello, world"  
hello, world
```

- ◆ Multiple arguments followed

```
#>echo "hello," "world"  
hello, world
```

I/O Command: printf

- ◆ Similar to “echo”, but with some formatting

```
#>printf "first name\tlast name\njack\t\tzheng\n"
first name      last name
jack            zheng
```

- ◆ Format controls reference: use man or visit
 - <http://wiki.bash-hackers.org/commands/builtin/printf>

I/O Command: read

- ◆ Accept user input from the keyboard and save it to a variable
 - Use -p option for input prompt

```
#> read -p "Enter something:" var1; echo "User input: $var1"  
Enter something: 1000  
User input: 1000
```

Variables

◆ Variable naming

- Variable names are case sensitive
- All-caps names typically suggest environment variables or variables read from global configuration files
- Local variables are all-lowercase with components separated by underscores

◆ Assignment

- All variables are of the string data type when assigned

```
#> var1="today"
```

no space around the = symbol

◆ Referencing variables

- Use the "\$"+variable name
- Use {} around variable name optionally

```
#>var1=1000; printf "User input: $var1\n"  
User input: 1000
```

Double and Single Quotes

- ◆ The shell treats strings enclosed in single and double quotes similarly, except that double-quoted strings are subject to globbing (the expansion of filename-matching metacharacters such as * and ?) and variable expansion.

- Example

```
#>mylang="Pennsylvania Dutch"  
#>echo "I speak ${mylang}."  
I speak Pennsylvania Dutch.  
#>echo 'I speak ${mylang}.'  
I speak ${mylang}.
```

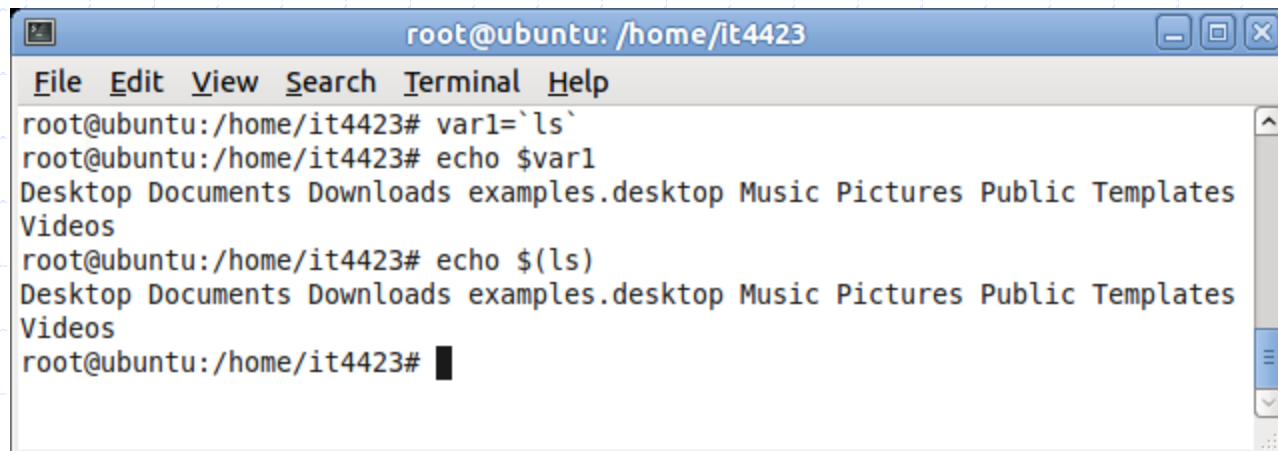
Capture Command Output

- Back quotes (or back-ticks), are similar to double quotes, but they have the additional effect of executing the contents of the string as a shell command and replacing the string with the command's output.

```
#>echo "There are `wc -l /etc/passwd` lines in the passwd file."  
There are 28 lines in the passwd file.
```

◆ \$()

- Another form of command substitution



```
root@ubuntu: /home/it4423  
File Edit View Search Terminal Help  
root@ubuntu:/home/it4423# var1=`ls`  
root@ubuntu:/home/it4423# echo $var1  
Desktop Documents Downloads examples.desktop Music Pictures Public Templates  
Videos  
root@ubuntu:/home/it4423# echo $(ls)  
Desktop Documents Downloads examples.desktop Music Pictures Public Templates  
Videos  
root@ubuntu:/home/it4423# █
```

Summary

◆ Key concepts

- Command-line scripting
- I/O channels: stdin, stdout, stderr
- Pipe
- Environmental variable

◆ Key skills: write simple command-line shell scripts utilizing the following elements

- Channel operators: > < >> 2> |
- Variables
- Shell builtin commands
 - ◆ echo, printf, read, bash

Good Readings and Resources

- ◆ A quick guide to writing scripts using the bash shell
 - <http://www.panix.com/~elflord/unix/bash-tute.html>