Administration of Operating Systems

DO2003

http://www.hh.se/do2003

The Bourne Again Shell (bash)

Lecture Overview

- At the end of the lecture students are able to:
  - Use bash
  - Create and execute shell scripts
  - Control jobs
  - Create variables, functions, aliases

The Bourne Again Shell (Bash)

Bourne Again Shell (bash)

- Is both a
  - Command interpreter
    - Executes commands
  - Programming language
    - Executes commands from files called shell scripts
- Bash is the default shell in GNU/Linux
  - `/bin/bash`

What is a shell script?

- A shell script is a text file containing shell commands
  - Might contain also control structures
- The shell interprets and executes the commands in a shell script, one after another.
- Some scripts will run at the startup for self initialization
- Others will be started by the user for maintenance, administration, or repair work purposes
- Functionality
  - Read data
  - Process data
  - Produce results

```
# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or ~/.bash_login
# exists.
# see /usr/share/doc/bash/examples/startup-files for examples.
# the files are located in the bash-doc package.

# the default umask is set in /etc/profile; for setting the umask
# for ssh logins, install and configure the libpam-umask package.
#umask 022

# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi

# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi
```

Startup files

- When a shell starts, it runs startup files to initialize itself
  - Login Shells
    - `/etc/profile`: system-wide initialization
    - `~/.bash_profile`: override `/etc/profile`
    - `~/.bash_login`: substitute for `~/.bash_profile`
    - `~/.profile`: substitute for `~/.bash_profile`
    - `~/.bash_logout`: executet at logout
  - Interactive nonlogin Shell
    - `/etc/bashrc`: system-wide aliases and functions
    - `~/.bashrc`: personalization of the user's shell environment

Running a shell script

- Startup files
  - Logout and login

```
$ . Filename [arguments]
```

- Otherwise
  1. `$ Filename [arguments]`
  2. `$ ./Filename [arguments]`
  3. `$ bash Filename [arguments]`
- 1 requires that the working directory be in the PATH variable
- 1 and 2 require that you have execute and read permission for the script file
- 3 requires that you have read permission for script file

Simple scripts

Script 1

```
$ cat > whoson
#Display date and  users logged in
date
who
$ whoson
Whoson: command not found
$ ./whoson
-bash: ./whoson: Permission denied
$ chmod u+x whoson
$ ./whoson
Sun Nov  6 01:58:29 PST 2011
ide      tty1     2011-11-05 23:43
$ dash whoson
Sun Nov  6 01:58:31 PST 2011
ide      tty1     2011-11-05 23:43
$
```

Script 2

```
$ cat > whoson2
#!/bin/bash
echo “Today’s date and users \
logged in”
(Date ; who)
$ dash whoson2
Today’s date and users logged in
Sun Nov  6 01:59:23 PST 2011
ide      tty1     2011-11-05 23:43
$ ./whoson2
-bash: ./whoson2: Permission denied
$ chmod u+x whoson
$ ./whoson2
Today’s date and users logged in
Sun Nov  6 01:59:00 PST 2011
ide      tty1     2011-11-05 23:43
```

Simple scripts II

```
$ cat > printArg.sh
#!/bin/bash
echo “The script has $# arguments.”
echo $0

$ cat > srm
#!/bin/bash
# Safe Remove
backup_path=/tmp
cp $1 $backup_path
echo $1 is stored in $backup_path
rm $1
```

Job control

- One or more commands connected by pipes
- Suspend a job
  - Pressing the suspend key, typically CONTROL-Z
- Bring a job running in the background into the foreground
  - fg
- Put a foreground job into the background
  - bg
- To see which jobs are running or suspended
  - jobs

Job control II

```
$ cat > script2.sh
#!/bin/bash
sleep 15&
$ chmod 700 script2.sh
$ ./script2.sh& ls& ps& jobs&
```

- How many jobs have been created?
- How many child processes have been forked?

- What about

```
$ (./script2.sh ; ls)& ps& jobs&
```

Shell parameters

- A parameter stores values
- A variable is a parameter denoted by a name.
- Positional Parameters
  - Command-line arguments

    `$#; $1,$2,…, $9; $?`
- Special Parameters
  - Parameters denoted by special characters and are not modifiable
- Users can define *User-created variables*

  `VARIABLE=value`
- The shell has *keyword variables*

User-created variables

- Must start with **letter** or **underscore** only
- You can change its value
- You can assign attributes to variables
  - Read-only (not removable), make it global, integer type
- You can remove a variable

```
$ declare myName='Wagner'
$echo myName
myName
$echo "$myName"
Wagner
$echo '$myName'
myName
```

```
$ declare -x myName="Wagner"
$ myAge=31
readonly age
$echo $myName
Wagner
$ unset myName
echo $myName
$
```

Keyword shell variables

- The shell creates and initialize keyword variables when it starts

```
$ echo $HOME
/home/ide
$ echo $PATH
/usr/local/sbin:/usr/local/bin:…
$ echo $LANG
en_US:UTF-8
$ echo $HISTFILE
/home/ide/.bash_history
```

Special characters I

- ; Separates commands
- ( ) Groups commands for execution by a subshell or identifies a function
- (( )) Expands an arithmetic expression
- & Executes a command in the background
- | Sends stdout of the preceding command to stdin of the following command
- > Redirects standard output
- >> Appends standard output
- < Redirects standard input

Special characters II

- * Any string of zero or more characters in an ambiguous file reference
- ? Any single character in an ambiguous file reference (page 256)
- \ Quotes the following character
- ' Quotes a string, preventing all substitution
- " Quotes a string, allowing only variable and command substitution
- ‘...‘ Performs command substitution
- [ ] Character class in an ambiguous file reference

Special characters III

- $ References a variable
- . (dot builtin) Executes a command
- # Begins a comment
- { } Surrounds the contents of a function
- && (Boolean AND) Executes command on right only if command on left succeeds
- || (Boolean OR) Executes command on right only if command on left fails
- ! (Boolean NOT) Reverses exit status of a command
- $() Performs command substitution
- [ ] Evaluates an arithmetic expression

Process

- A process is the execution of a command by the Linux kernel
  - If a command is not built in the shell (`cd`, `echo`, …), the shell forks a new (child) process to execute the command
  - The shell waits, sleep state, for the child process to finish
- Each process has a PID number (`ps -f`)
- Processes can run in the background (`&`)
  - The shell does not wait the process to finish
- When a process terminates, it returns its exit status (`$?`) to its parent process
  - 0 = success
  - Otherwise, failure

History

- The shell provides access to a list of commands previously typed, i.e., the *command history*
- From `~/.bashrc`

```
HISTCONTROL=ignoredups:ignorespace
HISTSIZE=1000
HISTFILESIZE=2000
```

- Bash assigns a event number to each issued command line

```
$ history | tail
…
416 vi ~/.bashrc
417 history|tail
```

Displaying and reexecuting commands

- Displaying the history list

  fc -l [first [last]]

  fc -l 5 10

  fc - echo vi

  - Reexecuting
    - fc –s 415
    - !415
- Event designators

Designator	Meaning
!	Starts a history event
!!	The previous command
!n	Command number n in the history list
!string	The most recent command line that started with string

Aliases

- Is a name that the shell translates into another name or (complex) command
- Aliases allow you to define new commands by substituting a string for the first token of a simple command

```
alias [name[=value]]
$ alias la=‘ls -a’
$ alias rm=‘rm -i’
```

  - To list active aliases

```
$ alias
```

Functions

- The shell allow user to define shell functions
- It's a way to group commands using a single name for the group.
- Functions are executed in the same was as a command
- Unlike a shell script, a function is parsed prior to being stored in memory
  - Run faster than shell scripts
- A function can be defined on the command line or within a shell script

```
[function] name ()  { commands }
```

- **Functions may not be empty**

Function example I

```
$ function whoson()
> {
> date
> echo “Users currently logged on”
> who
>}
$ whoson
Sun Nov   6 01:58:29 PST 2011
Users currently logged on
ide  tty1 2011-11-06 13:43
$
```

Function example II

```
$ add()
> {
> echo $(($1+$2))
> }
$ add 3 5
8
$ add() { echo $(($1+$2)) };
$ add 3 5
8
$
```

Function example III

- Compacted into a single line
  - $ myFunction () { echo "This is a function"; echo; }
- Many functions in a script file

```
$ cat > script1.sh
f1 ()
{
  echo “Calling \"f2\" from within \"f1\"."
  f2
}
f2 () { echo "Function \"f2\"."; }
```

Controlling bash

- The `set` and `shopt` builtins allows shell customization by turning features on and off
  - `set -o [argument]`
    - Turns on the feature indicated by the argument
    - If the argument is not indicated, it lists all features controlled by set
  - `set +o [argument]`
    - Turns off the feature indicated by the argument
    - If the argument is not indicated, it lists how you can switch the state of all features controlled by set

```
set -o noclobber
```

Command-line expansion

- After splitting the a command line into tokens, the bash may replace some words with expanded text
- Brace expansion

```
$ ls
script1.sh     script2.sh     script3.sh
script4.sh
$ chmod 700 script{1,2,3}.sh
```

- Tilde expansion
  - `~` is equal to the value of `$HOME`
  - `~/Docs` is equal to `$HOME/Docs`

Other Command-line expansions

- Parameter and variable expansion

```
$ myName='Wagner'
$ echo ${myName:='Unknown'}
Wagner
$ unset myname
$ echo ${myName:='Unknown'}
Unknown
```

- Quotation marks
  - `$ echo "$myName"` vs. `$ echo '$myName'`

Other Command-line expansions

- Arithmetic expansion

  $(($1+$2))

Operator	Meaning
VAR++, VAR--	variable post-increment and post-decrement
++VAR, --VAR	variable pre-increment and pre-decrement
! and ~	logical and bitwise negation
**	exponentiation
+, -, *, /, %	addition, subtraction, multiplication, division, remainder
<< and >>	left and right bitwise shifts
<=, >=, < , >	comparison operators
==, !=	equality and inequality
|, &, ^	bitwise AND, bitwise OR, bitwise exclusive OR
||, &&	logical OR, logical AND
expr ? expr : expr	conditional evaluation
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^= , |=	assignments
,	separator between expressions