More Scripting

A modification of Don Towsley's file which has been downloaded from the net a long-long time ago. Sorry...

Rev 2014-02-24-03

Commands for programmers

– man	Man pages
– time	How long your program took to run
– date	print out current date/time
– test	Compare values, existence of files, etc
– tee	Replicate output to one or more files
– diff	Report differences between two files
– sdiff	Report differences side-by-side
– wc	Show number of lines, words in a file
– sort	Sort a file line by line
– gzip	Compress a file
– gunzip	Uncompress it

Administrative Tools

- `wc`
  - count the number of characters, words and lines
- `cat`
  - display the contents of a file or join files
- `more` and `less`
  - Display the contents of a file a page at a time
- `head`
  - display the first few lines of a file
- `tail`
  - Display the last few lines of a file

- `sort`
  - sort the content of a file into order
- `uniq`
  - Remove duplicate lines from a file
- `cut`
  - remove columns of characters from a file
- `paste`
  - join columns of files together
- `tr`
  - translate specific characters
- `split`
  - split files evenly

Job control

- Start a background process:
  - program1 &
  - program1
    Hit CTRL-Z
    bg
- Where did it go?
  - jobs
  - ps
- Terminate the job: kill it
  - kill %*jobid*
  - kill *pid*
- Bring it back into the foreground
  - fg %1
- Start a job in the future
  - at

Kill: send signals

- When to use
  - Terminate a process, sending TERM
  - Send any signals.
  - Syntax

    kill [-signal] pid

    Note: pid = -1 may mean all process except system processes and the current shell. See man pid for more options.
  - Kill pid
    - Can be caught, blocked and ignored
  - kill –9 pid
    - Guarantee the process die?

Pipelines

- Pipes take the output of the first program and feed that output into the input of the next program.
- Also sometimes known as “filters”.
- Examples:

```
last | less

last | grep ^root | less

last | grep ^root | cut -d -f 2 | less

grep “error” something.out | tail -1
```

Redirection: < >

- >&*filename* redirects the standard output and error to the file called *filename*:

  ```
  last | grep ^root  >& root-logins.txt
  less root-logins.txt
  ```
- >*filename* redirects just standard output
- Don’t Clobber me! By default, > will overwrite existing files, but you can turn this off using shell settings and/or environment variables.
- Appendicitis! You can append to existing files this way:
  - *sh: >>filename >&1*
  - *csh: >>&filename*
- Use < to redirect a file to a command’s *standard input*

  ```
  # cat calculation.txt
  (3+2)*8
  # bc < calculation.txt
  40
  ```
- Useful when a program does not already query the command line for files to read

Pipelining into awk

- Manipulate the output of another command
- Picking out the columns

  Example:

  - List the users that run dooms.

    ```
    $ps –ef | grep “[d]oom” | awk ‘{print $1}’
    ```

  - Create a file to store the users that run dooms, include the data, cpu time

    ```
    $ (date ; ps –ef | grep “[d]oom” | awk
      ‘{print $1 “ [ “ $7 “]” }’ | sort | uniq)
      >> doomed.users
    ```

Conditional Execution

- program1 && program2
  - Program 2 will execute if and only if program1 exited with a 0 status
  - Example:
    - project1 && echo “Project1 Finished correctly!”
- program1 || program2
  - Program 2 will execute if and only if program1 exited with a *non*-0 status
  - Example:
    - project1 || echo “Project1 FAILED to complete!”
- Exit a script with an error:
    - exit 1

FIND

- Find locates files having certain characteristics on where you tell it to look.
- Basic syntax

  #find starting-dir(s) criteria-and-action
- Matching criteria
- Action
  - What to do with the files matches all the criteria

-atime n	File was last accessed n days ago
-mtime n	File was last modified exactly n days ago
-newer file	File was modified more recently
-size n	File is exactly n 512-byte blocks long
-type c	Specifies file typeL f, d
-name nam	The filename is nam
-perm p	The file's access mode is p
-user usr	The file's owner is usr
-group grp	The file's group owner is grp
-nouser	The file's owner is not listed
-nogroup	The file's group owner is not listed

- Use +, - to indicate more than, less than
  - -mtime +7 last modified more than 7 days ago
  - -atime –2 last accessed less than 2 days ago
  - -size +100 larger than 50k
- Use wildcards with –name option
  - -name “*.dat”
- Join more condition together
  - Or relation -o

    \(-atime +7 –o –mtime +30 \)
  - Not relation !

    ! –name gold.dat –name *.dat

- Check for a specific access mode with –perm
  - Exact permission
    - -perm 75
  - At least permission with “‘-” sign
    - -perm –002   world writable
    - -perm –4000   SUID access is set
    - -perm –2000   SGID access is set

option	**Meaning**
-print	Display pathname of matching file
-ls	Display long directory listing for matching files
-exec cmd	Execute command on file
-ok cmd	Prompt before executing command on file
-xdev	Restrict the search to the file system of the starting directory
-prune	Don't descend into directories encountered

- Default is –print
  - Example: $ find . –name *.c -print
- -exec and –ok must end with \;
- {} may be used in commands as a placeholder for the pathname of each found file.
  - -exec rm –f {} \;

FIND (examples):

- The usage of find for administration
  - Monitoring disk usage
  - Locating file that pose potential security problems
  - Performing recursive operations
- Example:

```
$find /chem -size +2048 -mtime +30 -exec ls -l {} \;
$find /chem -size +2048 \( -mtime +30 -o -atime +120 \) -ls
$find / \( -perm -2000 -o -perm -4000\) -print | diff - files.secure
$find /chem -name '*.c' -exec mv {} /chem1/src \;
```

Shell programming

- When you create a shell script using a editor
  - does it have execute permission typically?
  - Example

```
$ ./test
./test: Permission denied.
$ ls -l test
-rw------- 1 user user 22Jan08 test
$ chmod +x test
$ ./test
this is a test
```

Bourne Shell Programming

- Control structures
  - if … then
  - for … in
  - while
  - until
  - case
  - break and continue

if … then

- Structure

```
if test-command
   then
      commands
fi
```

Example:

```
if test  "$word1"  =  "$word2"
     then
          echo "Match"
 fi
```

test

- Command test is a built-in command
- Syntax

```
test expression
[ expression ]
```

  - The test command evaluate an expression
  - Returns a condition code indicating that the expression is either true (0) or false (not 0)
- Argument
  - Expression contains one or more criteria
    - Logical AND operator to separate two criteria: -a
    - Logical OR operator to separate two criteria: -o
    - Negate any criterion: !
    - Group criteria with parentheses
  - Separate each element with a SPACE

Test Criteria

- Test Operator for integers: int1 relop int2

Relop	Description
-gt	Greater than
-ge	Greater than or equal to
-eq	Equal to
-ne	Not euqal to
-le	Less than or equal to
-lt	Less than

Exercise

- Create a shell script to check there is at least one parameter
  - Something like this:

```
...
if test $# -eq 0
   then
        echo “Supply at least one argument”
        exit  1
 fi
…
```

Test Criteria

- The test built-in options for files

Option	Test Performed on file
-d filename	Exists and is a directory file
-f filename	Exists and is a regular file
-r filename	Exists and it readable
-s filename	Exists and has a length greater than 0
-u filename	Exists and has setuid bit set
-w filename	Exists and it writable
-x filename	Exists and it is executable
… …	… …

Exercise

- Check weather or not the parameter is a non-zero readable file name
  - Continue with the previous script and add something like

```
if [ -r  “$filename” –a –s “$filename” ]
then
        … …
fi
```

Test Criteria

- String testing

Criteria	meaning
String	True if string is not the null string
-n string	True if string has a length greater than zero
-z string	True if string has a length of zero
String1 = string2	True if string1 is equal to string2
String1 != string2	True if string1 is not equal to string2

Exercise

- Check users confirmation
  - Frist, read user input

```
echo  -n “Please confirm: [Yes | No] “
read user_input
```

  - Then, compare it with standard answer ‘yes’

```
if  [  “$user_input”  =  Yes  ]
then
        echo “Thanks for your confirmation!”
fi
```

  - What will happen if no “” around $user_input and user just typed return?

if…then…else

- Structure

```
if test-command
  then
    commands
  else
    commands
fi
```

  - You can use semicolon (;) ends a command the same way a NEWLINE does.

```
if [ …  ]; then
   … …
fi

if [ 5 = 5 ]; then echo "equal"; fi
```

if…then…elif

- Structure

```
if test-command
   then
      commands
   elif test-command
     then
         commands
    .
    .
    .
     else
         commands
fi
```

Debugging Shell Scripts

- Display each command before it runs the command
  - Set the –x option for the current shell
    - $set –x
  - Use the –x to invoke the script
    - $sh –x command arguments
    - **`sh -x coba 1 2 3 4`**
  - Add the set command at the top of the script
    - set –x
- Then each command that the script executes is preceded by a plus sign (+)
  - Distinguish the output of trace from any output that the script produces
- Turn off the debug with set +x

for… in

- Structure

  for ***loop-index*** in ***argument_list***

  do

  commands

  done

  Example:

```
for  file in *
 do
        if [ -d "$file" ]; then
              echo $file
        fi
  done
```

for

- Structure

```
for loop-index
do
    commands
done
```

  - Automatically takes on the value of each of command line arguments, one at a time. Which implies

```
for arg in "$@"
```

while

- Structure

```
while test_command
do
    commands
done
```

Example:

```
while [ “$number” –lt 10 ]
do
        … …
        number=`expr $number + 1`
done
```

until

- Structure

```
until test_command
do
    commands
done
```

Example:

```
secretname=jenny
name=noname
until  [ "$name" = "$secretname" ]
do
        echo " Your guess: \c"
        read name
done
```

break and continue

- Interrupt for, while or until loop
- The break statement
  - transfer control to the statement AFTER the done statement
  - terminate execution of the loop
- The continue statement
  - Transfer control to the statement TO the done statement
  - Skip the test statements for the current iteration
  - Continues execution of the loop

Example:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
        if [ $index –le 3 ]; then
            echo continue
            continue
        fi
        echo $index
        if  [ $index –ge 8 ]; then
            echo “break”
            break
        fi
 done
```

case

- Structure

```
case  test_string in
pattern-1 )
        commands_1
        ;;
pattern-2 )
        commands_2
        ;;
… …
esac
```

- default case: catch all pattern

```
* )
```

case

- Special characters used in patterns

Pattern	Matches
*	Matches any string of characters.
?	Matches any single character.
[…]	Defines a character class. A hyphen specifies a range of characters
|	Separates alternative choices that satisfy a particular branch of the case structure

Example

```
#!/bin/sh
echo "\n Command MENU\n"
echo " a. Current data and time"
echo " b. Users currently logged in"
echo " c. Name of the working directory\n"
echo "Enter a,b, or c:  \c"
read answer
echo
case "$answer" in
   a)
        date
        ;;
   b)
        who
        ;;
   c)
        pwd
        ;;
    *)
        echo "There is no selection: $answer"
         ;;
esac
```

echo and read

- The backslash quoted characters in echo
  - \c suppress the new line
  - \n new line
  - \r return
  - \t tab
- Read
  - read variable1 [variable2 …]
    - Read one line of standard input
    - Assign each word to the corresponding variable, with the leftover words assigned to last variables
    - If only one variable is specified, the entire line will be assigned to that variable.

Built-in: exec

- Execute a command:
  - Syntax: exec command argument
  - Run a command without creating a new process
    - Quick start
    - Run a command in the environment of the original process
    - Exec does not return control to the original program
    - Exec can be the used only with the last command that you want to run in a script
    - Example, run the following command in your current shell, what will happen?

```
$exec who
```

Built-in: exec

- Redirect standard output, input or error of a shell script from within the script
  - exec < infile
  - exec > outfile 2> errfile

- Example:

```
sh-2.05b$ more redirect.sh
exec > /dev/tty
echo "this is a test of redirection"

sh-2.05b$ ./redirect.sh 1 > /dev/null 2 >& 1
this is a test of redirection
```

Catch a signal: builtin trap

- Built-in trap
  - Syntax: trap ‘commands’ signal-numbers
  - Shell executes the commands when it catches one of the signals
  - Then resumes executing the script where it left off.
    - Just capture the signal, not doing anything with it

      ```
      trap ‘ ‘ signal_number
      ```
  - Often used to clean up temp files
  - Signals

• SIGHUP	1	disconnect line
• SIGINT	2	control-c
• SIGKILL	9	kill with -9
• SIGTERM	15	default kill
• SIGSTP	24	control-z
• …		

Example

```
$ more inter
#!/bin/sh
trap 'echo PROGRAM INTERRUPTED' 2
while true
do
        echo "programming running."
        sleep 1
done
```

A partial list of built-in

bg, fg, jobs	job control
break, continue	change the loop
cd, pwd	working directory
echo, read	display/read
eval	scan + evaluate the command
exec	execute a program
exit	exit from current shell
export, unset	export/ remove a val or fun
test	compare arguments

A partial list of built-in

kill	sends a signal to a process or job
set	sets flag or argument
shift	promotes each command line argument
times	total times for the current shell
trap	traps a signal
type	show if command, build-in, or function
umask	file creation mask
wait	waits for a process to terminate.
ulimit	the value of one/more resource limits

functions

- A shell function is similar to a shell script
  - It stores a series of commands for execution at a later time.
  - The shell stores functions in the memory
  - Shell executes a shell function in the same shell that called it.
- Where to define
  - In .profile
  - In your script
  - Or in command line
- Remove a function
  - Use unset built-in

functions

- Syntax

```
function_name()
    {
        commands
    }
```

- Example:

```
sh-2.05b$ whoson()
> {
> date
> echo "users currently logged on"
> who
> }
sh-2.05b$ whoson
Tue Feb  1 23:28:44 EST 2005
users currently logged on
ruihong  :0          Jan 31 08:46
ruihong  pts/1       Jan 31 08:54 (:0.0)
ruihong  pts/2       Jan 31 09:02 (:0.0)
```

Example

```
sh-2.05b$ more .profile
setenv()
{
   if [ $# -eq 2 ]
   then
     eval $1=$2
     export $1
   else
     echo "usage: setenv NAME VALUE" 1>&2
   fi
}
sh-2.05b$. .profile
sh-2.05b$ setenv T_LIBRARY /usr/local/t
sh-2.05b$ echo $T_LIBRARY
/usr/local/t
```