Unix Scripts and Job Scheduling

Michael B. Spring
Department of Information Science and Telecommunications
University of Pittsburgh
spring@imap.pitt.edu
http://www.sis.pitt.edu/~spring

Overview

- Shell Scripts
  - Shell script basics
  - Variables in shell scripts
  - Korn shell arithmetic
  - Commands for scripts
  - Flow control, tests, and expressions
  - Making Scripts Friendlier
  - Functions
  - Pipes and Shell Scripts
  - Scripts with awk and/or sed
- Job Scheduling
  - bg and at
  - cron

Running a Shell Script

- First three forms spawn a new process, so new variable values are not left when you return
  - sh < filename – where sh is the name of a shell
    - does not allow arguments
  - sh filename
  - filename
    - Assumes directory in path
    - Assumes chmod +x filename
  - . filename
    - Does not spawn a new shell.
    - Changes to system variables impact the current shell
- you may exit a shell script by
  - Getting to the last line
  - Encountering an exit command
  - Executing a command that results in an error condition that causes an exit.

Structure of a Shell Script

- Basic structure
  - #! Program to execute script
  - # comment
  - Commands and structures
- Line continuation
  - | at the end of the line is an assumed continuation
  - \ at the end of a line is an explicit continuation
- # in a shell script indicates a comment to \n
- Back quotes in command cause immediate execution and substitution

Debugging a script

- Use the command set –x within a script
- You can also activate the following set options
  - -n read commands before executing them – for testing scripts
  - -u make it an error to reference a non existing file
  - -v print input as it is read
  - - disable the –x and –v commands
- Set the variable PS4 to some value that will help – e.g. ‘$LINENO: ‘

Calculations with expr

- Executes simple arithmetic operations
  - Expr 5 + 2 returns 7
  - Expr 7 + 9 / 2 returns 11 – order of operations
  - Spaces separating args and operators are required
- expr allows processing of string variables, e.g.:
  - var=`expr $var + n`
  - n.b. Korn shell allows more direct arithmetic
- Meta characters have to be escaped. These include (), * for multiplication, and > relational operator, and | and & in logical comparisons

Other Operations with expr

- expr arg1 rel_op arg2 does a relational comparison
  - The relational operators are =, !=, >, <, >=, <= -- <
  - return is either 0 for false or 1 if true
  - arg1 and arg2 can be string
- expr arg1 log_op agr2 does a logical comparison
  - arg1 | arg2 returns arg1 if it is true otherwise arg2
  - arg1 & arg2 returns arg1 if arg1 and arg2 are true else 0
- expr arg1 : arg2 allows regular pattern matching
  - The pattern is always matched from the beginning
  - If arg2 is in escaped ()'s, the string matched is printed, else the number of characters matched

Korn Shell Arithmetic (review)

- Assumes variables are defined as integers
- Generally, we will use the parenthetical form in scripts:
  - $((var=arith.expr.))
  - $((arith.expr))
- Generally we will explicitly use the $ preceding the variable -- although it can be omitted
- An example:
  - $(($1*($2+$3)))

Variables in Shell Scripts

- Variables are strings
- To include spaces in a variable, use quotes to construct it
  - var1="hi how are you"
- To output a variable without spaces around it, use curly braces
  - echo ${var1}withnospaces
- SHELL variables are normally caps
  - A variables must be exported to be available to a script
  - The exception is a variable defined on the line before the script invocation

Command Line Variables

- command line arguments
  - $0 is the command file
  - arguments are $1, $2, etc. through whatever
- they are expanded before being passed
- Special variables referring to command line arguments
  - $# tells you the number
  - $* refers to all command line arguments
- When the number of arguments is large, xarg can be used to pass them in batches

Handling Variables

- Quoting in a shell script aids in handling variables
  - “ “ -- $interpreted and ` ` executed
  - ‘ ‘ – nothing is interpreted or executed
- Null variables can be handled two ways
  - The set command has switches that can be set
    - Set –u == treat all undefined variables as errors
    - Set has a number of other useful switches
  - Variables may be checked using ${var:X}
    - ${var:-word} use word if var is not set or null – don’t change var
    - ${var:=word} sets var to word if it is not set or null
    - ${var:?word} exits printing word if var is not set or null
    - ${var:+word} substitutes word if var is set and non null

Commands for Scripts

- Shell script commands include
  - set
  - read
  - “Here” documents
  - print
  - shift
  - exit
  - trap

set

- set also has a number of options
  - -a automatically export variables that are set
  - -e exit immediately if a command fails (use with caution)
  - -k pass keyword arguments into the environment of a given command
  - -t exit after executing one command
  - -- says - is not an option indicator, i.e. –a would now be an argument not an option

Read and “here” documents

- read a line of input as in
  - read var
  - read <4 var (where 4 has been defined in an exec <4 file
- “here” documents
  - in a shell script, input can come from the script using the form
    - <<symbol
    - input
    - symbol
  - basically, it means read input for the command
  - reading stops when symbol is encountered

Example of a “here document”

```
# a stupid use of vi with a here file
vi -s $1 <<**cannedinput**
G
dd
dd
dd
:wq
**cannedinput**
```

print, shift, exit, and trap

- print
  - preferred over echo in shell scripts
  - the –n option suppresses line feeds
- shift
  - moves arguments down one and off list
  - does not replace $0
- exit
  - exits with the given error code
- trap
  - traps the indicated signals

An example of trap and shift

```
# trap, and in our case ignore ^C
trap 'print "dont hit control C, Im ignoring it"' 2
# a little while loop with a shift
while [[ -n $1  ]]
do
   echo $1
   sleep 2
   shift
done
```

Shell Script Flow Control

- Generally speaking, flow control uses some test as described above.

```
if sometest
    then
        some commands
    else
        some commands
fi
```

- A test is normally executed using some logical, relational, string, or numeric test

Tests

- The test command allows conditional execution based on file, string, arithmetic, and or logic tests
- test is used to evaluate an expression
  - If expr is true, test returns a zero exit status
  - If expr is false test returns a non-zero exit status
- [is an alias for test
  -] is defined for symmetry as the end of a test
  - The expr must be separated by spaces from [ ]
- test is used in if, while, and until structures
- There are more than 40 test conditions

File Tests

- -b block file
- -c character special file
- -d directory file
- -f ordinary file
- -g checks group id
- -h symbolic link
- -k is sticky bit set
- -L symbolic link
- -p named pipe
- -r readable
- -s bigger than 0 bytes
- -t is it a terminal device
- -u checks user id of file
- -w writeable
- -x executable

String, Logical, and Numeric Tests

- Strings
  - -n if string has a length greater than 0
  - -z if string is 0 length
  - s1 = s2 if string are equal
  - s1 != s2 if strings are not equal
- Numeric and Logical Tests
  - -eq -gt -ge -lt -ne -le numerical comparisons
  - ! -a -o are NOT, AND, and OR logical comparisons

Shell Script Control Structures

- Structures with a test
  - if [test] then y fi
  - if [test] then y else z fi
  - while [test] do y done
  - until [test] do y done
- Structures for sets/choices
  - for x in set do y done
  - case x in x1) y;; x2) z ;; *) dcommands ;; esac

if

- if [test] then {tcommands} fi
- if [test] then {tcommands} else {ecommands} fi
- if [test] then {tcommands} elif [test] then {tcommands} else {ecommands} fi
  - Commands braces are not required, but if used:
    - Braces must be surrounded by spaces
    - Commands must be ; terminated
  - Test brackets are optional, but if used must be surrounded by spaces

Sample if

```
if [ $# -lt 3 ]
then
   echo "three numeric arguments are
   required"
   exit;
fi
echo $(( $1*($2+$3) ))
```

while and until

- while
  - while test do commands done
- until
  - until test do commands done
  - like while except commands are done until test is true

Sample while

```
count=0;
while [ count -lt 5 ]
do
  count=`expr $count + 1`
  echo "Count = $count"
done
```

for

- for var in list do commands done
  - var is instantiated from list
  - list may be derived from backquoted command
  - list may be derived from a file metacharacters
  - list may be derived from a shell positional agumment variable

Sample for

```
for lfile in `ls t*.ksh`
do
  echo "****** $lfile ******"
  cat $lfile | nl
done
```

case

- The case structure executes one of several sets of commands based on the value of var.

```
case var in
    v1) commands;;
    v2) commands;;
    *) commands;;
esac
```

  - var is a variable that is normally quoted for protection
  - the values cannot be a regular expression, but may use filename metacharacters
    - * any number of characters
    - ? any character
    - [a-s] any character from range
  - values may be or'd using |

select

- Select uses the variable PS3 to create a prompt for the select structure
- The form is normally

```
PS3="A prompt string: "
Select var in a x "z space"
Do
        Case "$var" in
                a|x) commands;;
                "z space") commands;;
                *) commands;;
        Esac
Done
```

- To exit the loop, type ^D
- Return redraws the loop

Sample select

```
PS3="Make a choice (^D to end): "
select choice in choice1 "choice 2" exit
do
  case "$choice" in
        choice1) echo $choice;;
        "choice 2") echo $choice;;
        exit) echo $choice; break;;
        * ) echo $choice;;
  esac
done
echo "you chose $REPLY"
```

Sample Scripts

- All of our scripts should begin with something like this:

```
#!/bin/ksh
# the first line specifies the path to the shell
# the two lines below are for debugging
# PS4='$LINENO:   '
# set –x
```

- In working with a script, functions are defined before they are invoked

Scripts to find and list files

```
#!/bin/ksh
# the reviewfiles function would normally be defined here
printf "Please enter the term or RE you are looking for: "
read ST
FILES=`egrep -l $ST *.ksh`

if [ ${#FILES}  -gt 0 ]
then
    reviewfiles
else
    echo "No files found"
fi
```

Reviewfiles function

```
reviewfiles()
{
        PS3=“Files contain $ST, choose one(^D or 1 to exit): "
        STIME=$SECONDS
        select choice in "ENTER 1 TO EXIT THE LOOP" $FILES
        do
                case "$choice" in
                        "ENTER 1 TO EXIT THE LOOP")  break;;
                        * ) echo "You chose ${REPLY}. $choice";
                        cat $choice | nl;
                        FTIME=$SECONDS;
                        echo “Process took $(($FTIME-$STIME)) secs";;
                esac
        done
}
```

FTP Function(1)

```
# dfine the host as a variable for more flexibility
ftphost=sunfire2.sis.pitt.edu
# grab a password out of a carefully protected file
# consider a routine that would search for a password
   for $host
exec 4< ${HOME}/.ftppass
read -u4 mypass
# this could be read from a file as well
print -n "Enter your username for $ftphost:  "
read myname
```

FTP Function(2)

```
# prepare the local machine
# this could have been done from within ftp
cd ${HOME}/korn/ftpfolder
rm access_log.09*;
rm *.pl
rm sample.log
```

FTP Function(3)

```
# start an ftp session with prompting turned off
# use the "here file" construct to control ftp
ftp  -n $ftphost <<**ftpinput**
user $myname $mypass
hash
prompt
cd weblogs
mget access_log.09*
mget *.pl
get sample_log
**ftpinput**
```

FTP Function(4)

```
# output to a log file and the screen

print "`date`: downloaded `ls access_log.* |
   wc -l` log files" | tee -a work.log
print "`date`: downloaded `ls *.pl | wc -l` analysis files" |
   tee -a work.log
```

Job Scheduling

- Multiple jobs can be run in Unix interactively
- The can be grouped, piped, made conditional
- To run a job in the background, issue the command in the following form:
  - job&
- Alternatively, run the job normally and then:
  - ^Z to suspend the job
  - bg at the command prompt to move the job to the background

Process control commands

- nice – runs a command (with arguments) at a lower priority
  - nice –15 myscript
  - The default priority is 10
  - Higher numbers represent lower priority
- ps – lists processes giving their process id
- kill – stops a process
  - kill 23456 – kills the process with ID 23456
  - kill –9 is an absolute kill and should be used with caution

Job scheduling post logout

- nohup – allows a command to be run even if the user logs
  - nohup myscript&
- at – runs a command at a specified time
  - at 19:00 –m < cmndfile
  - Executes cmndfile at 7:00pm and sends mail when done
  - At –k –m –f xyz.ksh 7pm
  - Execute xyz.ksh @7pm using korn and send mail
- atq, atrm – atq check the queue and atrm removes a given scheduled job

Crontab

- crontab is a utility for managing the tables that the process “cron” consults for jobs that are run periodically
- crontab allows a user who has the right to add jobs to the system chronological tables
  - crontab –e allows the user to edit their entries
  - crontab –l allows a listing of current entries
  - crontab –r removes all entries for a given user
  - crontab file adds the entries in file to your crontab

Format of crontab entries

- A normal crontab entry looks as follows
  - Min Hour DoM MoY DoW command
  - 5 * * * * /usr/bin/setclk
  - This will run setclk at 5 minutes past the hour of every day, week, etc.
  - * means every possible value
  - Multiple values of one type can be set , separted with no space
  - 0,5,10,15,20,25,30,35,40,45,50,55 * * * * would run the command every five minutes

Allowable values

- Minute 0-59
- Hour 0-23
- Day of month 1-31
- Month of year 1-12
- Day of week 0-6 with 0 being Sunday