2057-7

First Workshop on Open Source and Internet Technology for Scientific Environment: with case studies from Environmental Monitoring

7 - 25 September 2009

Shell Programming

Paul Bartholdi

Observatoire de Geneve
Chemin des Maillettes 51
CH-1290 Sauverny
Switzerland

Shell programming

Paul Bartholdi

OpenSource Workshop - 2009 – ICTP - Trieste

Introduction

Shell scripts are very high level programs using the Unix facilities.

Simple examples

```
ls -ltR
ls -ltR | head -7
ls -ltR > ls-ltR

ls -l ls-ltR

ls -ltR > ls-ltR_`date +"%y.%m.%d_%H:%M:%S"`

ls -l ls-ltR* | tail +2 | head -1

ls -ltR          \
  | head -16     \
  | awk ' ! /^total/ {tot += $5 ; \
        printf ( "%9d %s\n", tot, $0 ) }'

alias rsum "awk '\! /^total/ {tot += "\$"\!* ; \
         printf ( "'"%9d %s\n"'", tot, "\$0" ) }'"

ls -ltR | head -16 | rsum 5
```

Question

So, What is Shell programming ?

Introduction (real)

A (shell) script is an executable text file containing unix commands, including redirections, pipes and other scripts.

- Essentially all Unix commands can be used ;
- Similarely, most constructs found in usual programming languages are also available for scripts. Newer ones (tcl, perl) incorporate facilities of sed, awk, grep etc. or use completely new concepts as objects as in `python` or `scsh` (based on `SCHEME`).
- Many dialects exist. The main ones are those based on the original bourn shell (sh, ksh, bash, zsh), and those based on C (tcsh, csh).
- All consider commands, executable (program and script), and files as basic elements.
- It is quite usual to use tcsh for interactive and sh (ksh) or bash for scripts.

Shell Syntax

To be executable, a file just needs the `+x` bit set (`chmod a+x+ <file>`).

To differentiate the dialects, the file should start with a pseudo-comment in the form :

```
    #!/bin/bash
```

or whatever other shell (including awk, perl etc) is used.

If this line ends with a `-x`, then every line executed is echoed. This is very useful for debugging.

A good site to visit : `http://www.shelldorado.com/`

Parameters

In most (all ?) shells, parameters are positional.

Inside a script, `$n` refers to the n'th parameter passed to the script.

`$0` is the name of the script itself

`$#` is the number of parameters passed

Example :

```
#!/bin/sh
echo "You executed the shell script $0"
echo "You passed $# parameters"
echo "The first parameter was $1"
exit 0
```

Comments and Continuation

Any character between `#` and the end-of-line is treated as a comment.

It can appear anywhere in the line

If the line ends with a `\` (no space after it), the next one is considered as a continuation.

The number of continuation lines is essentially unlimited.

It is a good habit to align vertically the `\` continuations.

Inside a line, the `\` is used to protect special characters from interpretation.

Quotes

Quotes

Three quotes are used : `' '` , `" "` and `` ` ` ``

Inside `' '` , no special character is interpreted.

Inside `" "` , `$` `` `...` `` `!` `\` are the only ones interpreted.

The string inside `` ` ` `` is replaced by the string resulting from the execution of the commands inside the `` ` ` ``.

Quote Examples

```
%
Test="NoGood"
echo 1. Test            # just ascii string   (Test)
echo 2. $Test           # $ interpreted       (NoGood)
echo 3. \$Test          # $ not interpreted   ($Test)
echo 4. \\$Test         # \ and $ interpreted (\NoGood)
echo 5. "$Test"         # $ interpreted       (NoGood)
echo 6. '$Test'         # $ not interpreted   ($Test)

Today=`date | tr " " "_"`

cp Ex7.java Ex7.java_`date +"%m%d-%H%M%S"`
```

- What is the content of the variable Today ?
- What is the name of the copy of Ex7.java ?

Variables

Variable can be defined inside a shell. Except if exported, they are local, not seen outside the shell.

Variable names are made of letters, digits and underscore only, starting with a letter or an underscore.

They are defined with a = (bash, sh) , without a space around the = sign, or read form the standard input :

```
Test="Order==$1" ;   read answer
```

and are used, as for parameters, with a `$` in front, for them to be replaced with their content.

It is purely text manipulation. They can appear inside a "word". It is then necessary to enclose the name of the variable with `{ }` to obviate any confusion.

```
if [ x${answer} == "xY" ] ; then
  SetPower $level
fi
```

Global Variables

bash/sh/ksh/... Variables are made global with the command

```
export variable_name
```

This can be on the line where the variable is defined, or anytime later. `NETWATCH=etherreal export etherreal`

The list of all global variables is obtained with `export` without parameter.

csh/tcsh Local variables a re defined with :

```
set VAR=value
```

While global variables are defined with :

```
setenv VAR value
```

The list of all global variables is obtained with `printenv`

Special Variables

A few special variables are always available. Assignment to them is not allowed.

`$0` is the name of the current script/executable,

`$#` is the number of parameters,

`$n` are the parameters passed to the script,

`$@` is the list of all parameters enclosed between " "

`$_` is the command line fully qualified name f the current script,

`$$` is the pid (process id number) of the current script,

`$?` is the status of the most recently executed foreground pipeline,

`$( command )` is the result of the command (same as ‘command‘).

Environment Variables

The system looks for programs in the directories defined in the variable `PATH` , and execute the first found.

In the same way, man looks in the directories defined in the variable `MANPATH` , and the loader in the `LD_LIBRARY_PATH` for libraries.

The directory names are separated with colon (" `:` ") characters.

To add a new directory :

```
PATH=${PATH}:new_dir      PATH=new_dir:$PATH
```

Remarks for PATH

Remarks for PATH

- A generous `PATH` is predefined for most Unix, but not available for cron.
- the current directory (.) is usually part of `PATH` . It is better to have it at the end to avoid replacing system commands.
- it is good to have all executables in `$HOME/bin` or `$HOME/scripts` , and to add these directory to `PATH` in the `.login` file.
- do the same for your local man pages.
- to see the full `PATH` , use

  ```
  echo $PATH | tr ":" "\n"   [ | sort | uniq -c ]
  ```
- to find where an executable is : `which my_program`
- to find all copies of an executable :

  ```
  whereis my_program
  ```

envv to modify PATH

This small program make it easy to add, change, move or remove entries in a `PATH` variables. No entry will appear more than once. It will work equally well in sh and csh.

Example to add my_dir in front of `PATH` :

```
eval `envv add PATH my_dir 1`
```

or if many entries are changed :

envv (2)

If many entries are changed, better use (.login, .profile) :

```
env SHELL=/usr/bin/csh envv << ---EOF--- > /tmp/env
set   EDITOR                crisp
move PATH                   /usr/bin 1
add   PATH                  /unige/java1.2/bin
add   PATH                  /home/system/bartho/bin 1
add   LD_LIBRARY_PATH           /usr/local/lib
move LD_LIBRARY_PATH            /usr/lib 1
del   MANPATH               /usr/local/lsf5/5.0/man
---EOF---

source  /tmp/envv.$$
/bin/rm /tmp/envv.$$
```

File Name Modifiers (csh)

Variable can be modified with the following modifiers :

`<variable name>:r`	suppress all the possible suffixes
`<variable name>:s/old/new/`	substitutes new for old.

More modifiers are available, see man pages.

Example :

```
foreach file (*.java)
  echo " $file --> $file:r "
  cp $file:r $file:r_org
end
```

Conditionals (bash, csh)

bash/sh/ksh/...

```
if  list  ; then
  commands
else
  commands
fi
```

csh/tcsh

```
if ( expression ) then
  commands
else
  commands
endif
```

Case (bash, csh)

Case (bash, csh)

bash/sh/ksh/...

```
case  word  in
  pattern1 ) commands ;;
  pattern2 ) commands ;;
  * ) commands ;;
esac
```

csh/tcsh

```
switch ( string )
case label1 :
  commands
breaksw
...
default:
  commands
endsw
```

Loops (bash, csh)

bash/sh/ksh/...

```
for var in list ; do
  commands
done
```

csh/tcsh

```
foreach var ( list )
  commands
end
```

Conditional loops (bash, csh)

bash/sh/ksh/...

```
while list ; do
  commands
done
```

csh/tcsh

```
while ( expression )
  commands
end
```

Conditional Expression (bash, ksh)

Conditionals test the status resulting from the execution of the list.

Simple conditional expression are done using `[ expression ]`, or the compound `[[ compound expression ]]`.

Compound expressions are made of simple expressions linked with logical operators (`||`, `&&`, `!`).

Binary operators are :

```
    == , != , < , > , -eq , -ne , -le , -gt , -ge
```

Unary operators

Unary operators are (see man pages for all of them) :

`-a file`	file exists (any type)
`-d file`	file exists and is a directory
`-f file`	file exists and is a regular file
`-s file`	file exists and is not empty
`-r file`	file exists and is readable
`-w file`	file exists and is writable
`-x file`	file exists and is executable

Examples of Conditionals (bash)

Testing some parameters of a file

```
if [ ! -a $1 ] ; then
  echo " file $1 does not exists"
else
  if [[ -f $1 && -w $1 ]] ; then
    echo " file $1 is a regular file"
    echo "         and is writable"
  else
    echo " file $1 is not a regular file"
    echo "          or is not writable"
  fi
fi
```

Examples of Conditionals 2 (bash)

Counting to one minute

```
i=0
date
while test $i -le 60 ; do
  case $(($i%10)) in
    0 )  j=$(($i/10)) ;  echo -n $j  ;;
    5 )  echo -n '+'  ;;
    * )  echo -n '.'  ;;
  esac
  i=$(($i+1))
  sleep 1
done
echo ' '
date

Wed Sep  2 17:10:47 CEST 2009
0....+....1....+....2....+....3....+....4....+..
Wed Sep  2 17:11:48 CEST 2009
```

Examples of Loops (bash)

Converting all tiff files into postscript ones

```
for file in *.tiff ; do
  psfile=${1%.tiff}.ps
  convert $1 $psfile
  echo "$1 converted into $psfile"
done
```

Simple list of password file

```
count=0
while read whole_line ; do
  UserName=`echo $whole_line | cut -d":" -f1 `
  echo "User $count has username $UserName"
  count=`expr $count + 1`
done < /etc/passwd
```

Trapping Signals (bash, sh)

It is possible to execute a command in the case a signal is sent to the script.

Syntax :

```
trap " command " list of signals
```

Example :

```
trap "\rm -f $TMP" 0 1 2 3 9 15
```

A signal is sent with the command `kill n` where n is the number or the name of the signal

List of Signals

List of Signals

The most useful signals are :

 0 `-EXIT` not a real signal, just passed to the script

 1 `-HUP` hangup

 2 `-INT` interrupt, also generated with `<ctrl>C`

 3 `-QUIT` quit, also generated with `<ctrl>[`, explicitly request a core dump

 9 `-KILL` brutal death, cannot be caught or ignored

10 `-BUS` bus error

11 `-SEGV` segmentation violation

List of Signals - 2

`13 -PIPE` pipeline without reader to terminate a writing process

`15 -TERM` to terminate a process gracefully

`16 -USR1` user defined

`17 -USR2` user defined

`23 -STOP` stop momentarily the process

`25 -CONT` restart a stooped process

Script file : KillMeAfter

Suppose we have a script that should not take more than a few seconds, but hangs sometimes...

Here is an example ;

```
#!/bin/sh
host=$1
/home/b/bartho/bin/KillMeAfter $$ 80 &
if /sbin/ping $h 1 > /dev/null ; then
  if /usr/bin/ssh -n $host "date" > /dev/null ; then
    /usr/bin/ssh -n $host "/usr/bin/checkpc -f; \
                           /etc/init.d/lpd restart"
  fi
fi
/home/b/bartho/bin/KillKillMeAfter $$
exit 0
```

Script file : KillMeAfter

A lot of precautions have been taken, but we cannot be sure that checkpc or lpd will not hang.

The two commands `KillMeAfter` and `KillKillMeAfter` will do the job

Script file : KillMeAfter (2)

`KillMeAfter` gets two parameters.

- The first is the pid of the calling script.
- The second is the max time allowed for the calling script.

`KillMeAfter` will go to sleep for that time, and when waken-up, kills the calling script if it still exists.

`KillKillMeAfter` is called at the end of the script, and kills the sleeping KillMeAfter script. It has one parameter, the calling script pid (necessary because we don’t want to kill the wrong KillMeAfter !)..

Script file : KillMeAfter (3)

```
#!/bin/sh
# called by some script, with pid as parameter $1,
# expected to kill it after $2 sec

# echo $0 : pid=$1
# echo $0 go to sleep for $2 sec
sleep $2

# echo $0 weak up
if  `ps -ef -o pid | egrep $1 > /dev/null ` ; then
   kill -9 $1
#    echo pid : $1 should be dead now
# else
#    echo pid : $1 was already killed
fi

exit 0
```

Script file : KillKillMeAfter

```
#!/bin/sh
# Kill the KillMeAfter started by pid $1
# Also kill the sleep started by KillMeAfter

GAWK=/bin/gawk

KMApid=`ps -ef | tr -s ' ' | egrep KillMeAfter \
  | $GAWK -v pid=$1 '$10 == pid {print $2} ' `

sleeppid=`ps -ef | tr -s ' ' | egrep sleep \
  | $GAWK -v pid=$KMApid '$3 == pid {print $2} ' `

if [ "X$KMApid" != "X" ] ; then
  kill -9 $KMApid $sleeppid  2> /dev/null
fi

exit 0
```

Script file : snapshot backup

The cost of storage is now almost the same for disks and tapes. The goal of the following script is to make rotating backups on disks in a very efficient way, both in time and disk space.

Header :

```
#!/bin/sh
# ----------------------------------------------------------------
# mikes handy rotating-file-system-snapshot utility
# ----------------------------------------------------------------
# this needs to be a lot more general, but the
# basic idea is it makes rotating backup-snapshots
# of /home whenever called
# ----------------------------------------------------------------
# Local adaptation by Paul Bartholdi - 2002/10/02
# ----------------------------------------------------------------

echo "***** Start      : `date`"
```

Script file : snapshot backup (Definitions)

Definitions :

```
# ------------- system commands used by this script
ID=/usr/bin/id
ECHO=/bin/echo
MOUNT=/bin/mount
RM=/bin/rm
MV=/bin/mv
CP=/bin/cp
TOUCH=/bin/touch
RSYNC=/usr/bin/rsync

# ------------- file locations --------------------
SNAP_ORG=/home/b/bartho
SNAP_RW=/tmp
EXCLUDES=/home/b/bartho/backup_exclude
INCLUDES=/home/b/bartho/backup_include
```

Script file : snapshot backup (exclude file)

Provision is made for lists of directories to be included or excluded from the backups.

Directories can be fully rooted (/home/...) starting with a / , or starting with the last element of `$SNAP_ORG` as defined above.

Here is the content of the `backup_exclude` file

```
bartho/YESTERDAY_FILES
bartho/public
bartho/public_html
bartho/sm2_4_7
```

Script file : snapshot backup (Initialization)

Who is running the script ?

```
# ------------- the script itself -----------------
# make sure we're running as root
#if [ `$ID -u` != 0 ] ; then
#   $ECHO "Sorry, must be root.  Exiting..."
#   exit
#fi
```

Should be uncommented !

Script file : snapshot backup (Rotating snapshots)

Rotating backups, deleting oldest one :

```
# step 1: delete the oldest snapshot, if it exists:
echo "----- Step 1     : `date`"
if [ -d $SNAP_RW/backup.3 ] ; then
  $RM -rf $SNAP_RW/backup.3
fi ;

# step 2: shift the middle snapshots(s) back by one
#         if they exist
echo "----- Step 2     : `date`"
if [ -d $SNAP_RW/backup.2 ] ; then
  $MV $SNAP_RW/backup.2 $SNAP_RW/backup.3
fi;
if [ -d $SNAP_RW/backup.1 ] ; then
  $MV $SNAP_RW/backup.1 $SNAP_RW/backup.2
fi;
```

Script file : snapshot backup (Changed files)

Copy backup 0 with hard-links, and make new backup 0 :

```
# step 3: make a hard-link-only (except for dirs)
#         copy of the latest snapshot...
echo "----- Step 3    : `date`"
if [ -d $SNAP_RW/backup.0 ] ; then
  $CP -al $SNAP_RW/backup.0 $SNAP_RW/backup.1
fi

# step 4: rsync from the system into the latest snap
# rsync is like cp --remove-destination by default,
# so the destination  is unlinked first.
# If it were not so, this would copy over the other
# snapshot(s) too!
echo "----- Step 4    : `date`"
$RSYNC -valH --delete --delete-excluded          \
       --exclude-from="$EXCLUDES"                \
       --include-from="$INCLUDES"                \
           $SNAP_ORG $SNAP_RW/backup.0
```

Script file : snapshot backup (Finishing)

Update time of backup 0 and close :

```
# step 5: update the mtime of backup.0 to reflect
#          the snapshot time
echo "----- Step 5     : `date`"
$TOUCH $SNAP_RW/backup.0

# and thats all.

echo "===== All done   : `date`"

exit 0
```

Dear Mouse. . .