ARSC

Shell Scripting

David Newman

(From slides by Tom Logan)

(from Slides from Kate Hedstrom & Don Bahls)

ARSC

Overview

- Variables
- Scripting Basics
- Bash Shell Scripts
- Other Scripting
- Advanced Command Line
- Appendix (C-Shell Scripts)

Shells

- **There are a number of different shells available.**
- **Shell family tree.**
  - csh -> tcsh
  - sh -> ksh -> bash
- **This lecture focuses on bash**
  - user friendliness of tcsh
  - scripting capabilities of ksh
- **Most if not all syntax shown here works with ksh as well.**

Bash

- **bash is the bourne-again shell.**
- **Similar syntax to sh and ksh.**
  - Includes new features that are not in sh or older versions of ksh
  - Flexible syntax allow most expressions to be done on a single line (if you want).
  - Supports functions.
- **Default shell on most Linux systems.**
- **Verbose mode (useful for debugging)**

```
#!/bin/bash -v
```

Command Line

- **Consider the following:**

```
sort -n file > file.sorted &
```

- **sort is a command in your $PATH**
- **"-n file" is passed to the sort command as arguments**
- **> and & are special**
- **& puts the job in the background - DON'T do this in batch scripts**

Environment Variables

- **csh/tcsh:**

```
setenv PAGER more
```

- **sh:**

```
PAGER=more
export PAGER
```

- **ksh:**

```
export PAGER=less
```

- **Standard practice is to use uppercase names**

Variables

- **Bash (and other shells) allow users to instantiate local or environment variables.**
- **Environment variables are accessible to child shells.**

```
#local variable
num=20
#environment variable
export LD_LIBRARY_PATH="/usr/local/bin"
```

Environment Variables

- **The environment variable `PATH` defines a colon delimited list of directories where the shell (and other processes) should look for executables.**
- **At ARSC we use environment variables to define storage areas:**

  e.g.

```
cd $ARCHIVE
ls $CENTER
```

The $PATH

- Environment variable containing a list of directories to search for commands
- Order is important - takes first one
- Some commands are built into the shell, for instance echo is built into csh. There is also a /usr/bin/echo for shells which don't have echo built in.
- Can give the full path of commands to get a specific one: /usr/local/bin/patch or /usr/bin/patch
- Putting "." (current directory) in your path is controversial - put it at the end if you do

Setting the Environment

- **We often want the same environment variables to be set every time we log in**
- **For sh/ksh, set in .profile**
  - Can reload it with ". .profile"
- **For csh/tcsh, set in .cshrc**
  - Can reload it with "source .cshrc"

More Environment Variables

- Some have standard names, such as HOME, PATH, PRINTER, EDITOR
- Some programs are expecting environment variables to be set, for instance graphics programs: NCARG_ROOT, GMTHOME, QTDIR, MATLABPATH
- Programs can read the environment through the getenv function call

Shell Variables

- **Can be lowercase (case sensitive):**

```
name=Harry
echo $name
```

- **Quote for embedded spaces:**

```
longname='Harry Smith'
```

- **No spaces on either side of equals**

Scripting Basics

- **Scripts**
  - Usually executable

    e.g.

    ```
    chmod 700 myscript
    ```

    but don't necessarily need to be.

    ```
    ksh myscript
    ```

  - Should have the shell on the first line.

    e.g.

    ```
    #!/bin/ksh
    ```

Basics (continued)

- Be aware that following may work sometimes but are not portable! Don't write your scripts this way.
  - not specify a shell at the beginning of an executable script. **(BAD!)**
  - spaces between the "#!" and the shell.

    `#! /bin/sh` **(BAD?)**
  - Skipping the PATH to the script
  - `#!csh` **(BAD!)**

More on “#!”

- When you run a script interactively the program (i.e. shell) listed in the “#!” statement is started as child process of your login shell. It gets a copy of all of the environment variables set for the parent shell.
- aliases and functions are NOT inherited from the parent shell!

Integer Math

- **There are a few different ways to do math operations.**
- *var* = $((*expression*))

  e.g.

  ```
  x=$(( $y * 2 + 1 ))
  ```
- let *var* = (*expression*)

  e.g.

  ```
  let x=( $y * 2 + 1 )
  ```

If Statements (sh/ksh)

- **"if" has several flavors, including optional else and elif parts:**

```
if [ "$1" = south ]
then
    echo Going south
elif [ "$1" = north ]
then
    echo Going north
else
    echo Going east-west
fi
```

More on “if”

- **The example**

```
if [ -d /usr ]
```

- **Can be written**

```
if test -d /usr
```

- **test (or []) is testing the result of something**
- **An executable will return an error code and not need the test**

```
if hostname
```

Logical Operators

`&&` **logical and**

`||` **logical or**

`-a` **logical and (only within `[]`)**

`-o` **logical or (only within `[]`)**

`!` **logical negation**

`&&` **only performs second operation if the first succeeds (returns 0)**

`||` **only performs the second operation if the first operation failes (returns a non-zero value).**

File Tests

- `-d` *`val`* **val is a directory**
- `-e` *`val`* **val exists**
- `-f` *`val`* **val is a regular file (not a link or directory)**
- `-r` *`val`* **val is readable by user**
- `-w` *`val`* **val is writeable by user**
- `-x` *`val`* **val is executable by user**
- `f1 -nt f2` **f1 is newer than f2 *.**
- `f1 -ot f2` **f1 is older than f2 *.**

File Tests

- **Example - checking for writable file:**

```
if [ -w myfile ]
then
        ls >> myfile
fi
```

Logic Examples

```
if [ -d $f ] && [ -w $f ]; then
...

if [ -d $f -a -w $f ]; then
...

if [ -d $f -a \( -w $f -o -x $f \) ]
then
...
```

Other Operators

- `-z val` **string is zero length**
- `strA = strB` **are strings the same**
- `strA != strB` **are strings different**
- **Arithmetic Operators**

  `-eq` (equal), `-ne` (not equal)

  `-lt` (less than), `-gt` (greater than)

Warning about `[ ... ]`

- **If you have a variable that might not be set, put it in double quotes:**

```
if [ -z "$f" ]; then
    ...
fi
```

- **or use "`[[`" and "`]]`"**

```
if [[ -z $f ]]; then
```

Example of bad behavior from [

```
# show that f is not defined.
mg56 % echo $f

# this is bad!  The directory "$f" doesn't exist.
mg56 % if [ -d $f ]; then echo Hello; fi
Hello
# this is OK.
mg56 % if [ -d "$f" ]; then echo Hello; fi
# so is this.
mg56 % if [[ -d $f ]]; then echo Hello; fi
# if "$f" is defined we don't have this problem:
mg56 % f=.
mg56 % if [[ -d $f ]]; then echo Hello; fi
Hello
mg56 % if [ -d "$f" ]; then echo Hello; fi
Hello
```

Loops

- **sh/ksh**

```
for num in 42 66 210 13
do
    echo $num
done
```

- **csh/tcsh**

```
foreach lib (lib*)
    nm $lib | grep rand
    echo $lib done
end
```

for

```
for name in list; do
     #do something
done

for f in /usr/bin/*; do
  if [ ! -L $f -o ! -d $f ]; then
    echo $f
  fi
done
```

while

```
while [ logical-expression ];
do
    #do something
done
```

select

- **Simple command line parsing code blurb.**

```
case $arg in
  -a )
        #do something ;;
  -b )
        #do something else ;;
  * )
        #match everything else ;;
esac
```

Command Line Arguments

- **The variable `$0` has the name of the executable being run. `$1`-`$9` have the 1st thru 9th command line arguments.**
- `$#` **has the number of args**
- `$*` **can access all args (beyond 10)**
- `shift` **allows you to move an argument forward in the list.**

getopts

- **If you want to have a script accept command line arguments (e.g. “-f”), use getopts.**

```
while getopts “fg:” opt; do
 case $opt in
        f )  echo "-f is $OPTARG" ;;
        g )  echo "-g is $OPTARG" ;;
        \? ) echo "Usage: ..."
             exit 1
             ;;
    esac
done
# this allows “cmd -f -g arg” or “cmd -fg arg”
shift $(($OPTIND - 1 ))
```

getopts continued

- **The string** `"fg:"` **tells the script to look for** `"-f"` **and/or** `"-g val"`
- **The** `":"` **tells getopts that the preceding value must have an option.**
- `OPTARG` **and** `OPTIND` **are set by getopts**

Functions and Aliases

- **Simplify repeated tasks.**
- **However, aliases and functions are not inherited by child shells.**
- **You can source a file from within a script to get the functions and aliases from that file**
- **e.g.**

```
. ~/.mystuff
```

- **“. “ is source in bash and ksh**

Example Functions and Aliases

```
alias ll="ls -l"
function foo
{
    for l in $*; do echo $l; done
}
```

- **You can ignore an alias, function or built-in command by escaping the name.**
- **e.g.**

```
% \ls
```

Error Handling

- As previously mentioned normal convention is that programs exit with a non-zero value if they exit in error.
- We can use this to our advantage:
- e.g.

```
mv myfile $ARCHIVE || exit 1
```

- The exit value of the last command is stored in the variable "`$?`".
- We can give a more meaningful error message!

More Error Handling

- **A function can improve this alot.**

```
function printError
{
# $1 (optional) is an error message to print.
exitval=$?
if [ $exitval -ne 0 ]; then
    if [ ! -z "$1" ]; then
        echo "Error: $1"
    fi
    exit $exitval
fi
}

mv myfile $ARCHIVE || printError "mv myfile failed"
```

Quoting

- **The shell interprets these characters in a special way:**

```
# * ? \ [] {} () < > “ ‘ ` | ^ & ; $
```

- **Double quotes protect some, but allow $variable substitution:**

```
echo $PATH
echo “$PATH”
echo ‘$PATH’
echo \$PATH
```

Quoting Continued

- **Be aware of quoting.**
  - Variables are not expanded when within single quotes ‘’, but are in double quotes “”.

```
% echo "$PATH"
/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin:/usr/X11R6/bin

% echo '$PATH'
$PATH
```

  - Variables can also be escaped with “\”

```
% echo ”\$PATH"
$PATH
```

Subshells

- **Back ticks start a subshell and return the value**

```
% ls -l `which cat`
-r-xr-xr-x   1 root  wheel  14380 Mar 20  2005 /bin/cat
```

- **The $(...) operation works the same.**

```
% ls -l $(which cat)
-r-xr-xr-x   1 root  wheel  14380 Mar 20  2005 /bin/cat
```

Back Quotes

- **Can save the results of commands into a variable:**

```
pwd=`pwd`
lines=`cat /etc/passwd | wc -l`
echo $pwd
echo $lines
```

Shell Special Characters

- `*` **matches anything**
- `?` **matches on single character**
- `[a-z]` **matches a range of characters**
- `[^a-z]` **negation of the previous.**
- `{str1,str2}` **matches** `str1` **or** `str2`

Pipes and Redirection

- **Pipes allow you to send the "stdout" from one command to the "stdin" of another command.**

```
ls | more
```

- **Redirection allows you to send output to a file or input from a file.**

```
# look for the work fred in the file friends
grep -i fred < friends
# redirect the output of ls to a file called ls.out
ls > ls.out
# concatenate the output of ls to the file ls.out
ls >> ls.out
```

Tieing Output / Redirecting Stderr

- **Stdout can tied to stderr.**

```
echo "Error: " 1>&2
```

- **Stderr can tied to stdout.**

```
somecmd 2>&1
```

- **Redirecting Stderr.**

```
find . -name \*.out 2> /dev/null
```

Other Scripting Languages

- **If you end up needing to do more complicated operations. Consider a more powerful scripting language.**
  - python
  - perl
  - tcl/tk
  - ruby

Advantages

- **Languages like python have a large number of modules which come with the package.**
- **Python also have:**
  - Good integration with C/C++ and Fortran
  - Scientific Packages (numpy / scipy) give matlab like functionality.
  - Regular expressions for parsing files.

References

- **Linux in a Nutshell - O'Reilly (bash and tcsh)**
- **UNIX in a Nutshell - O'Reilly (csh, sh and ksh)**
- **Learning the bash shell - O'Reilly**

Appendix

ARSC

C-Shell

- based on C programming language syntax.
- tcsh has a bit more functionality if you want it.

Setting variables

- **Local variables (not available to child processes)**
  - `set v=0`
- **Environment variables available to child processes**
  - `setenv NCARG_ROOT /usr/local/pkg/ncl/ncl-4.2.0-a33/`
- **Arrays (Warning to C programmers first element of the array is 1 not 0!)**
  - `set arr=(“a” ”b” “c”)`
  - `echo ${arr[1]}`
  - `#echos a`

Arithmetic Operations

```
# set value of v to 0
set v=0
# set v to v + 1 (be careful about spacing!)
@ v=($v + 1)
#x     x x    here's where the spaces need to be.
# value of v is 1

@ v=($v * 2)
#x     x x    here's where the spaces need to be.
# value of v is 2
```

If

```
if ( ! -e $ARCHIVE/myresults ) then
   mkdir $ARCHIVE/myresults
endif

if ( -f ~/.myaliases ) then
   source ~/.myaliases
else
   echo "Warning ~/.myaliases not found"
endif
```

Tests

- `-d foo` **(is** `foo` **is a directory?)**
- `-e foo` **(does** `foo` **exist?)**
- `-f foo` **(is** `foo` **a regular file?)**
- `-l foo` **(is** `foo` **a symbolic link?)**
- `-o foo` **(is** `foo` **owned me?)**

- **tcsh has some additional tests which could be useful (groups** `-G foo`**, access time** `-A foo`**, permissions** `-P foo` **and more!)**

Logical Operators

- `&&` **logical and, performs second operation only if the first succeeds.**

```
mv foo $ARCHIVE && ls -la $ARCHIVE/foo
```

- `||` **logical or, performs the second operation only if the first fails.**

```
mv foo $ARCHIVE || echo $status
```

Error Checking

- **Programs exiting in error return a non-zero value.**
- **Programs that complete successfully return 0.**
- **This lets us test for errors.**
- **The variable `$status` (csh/tcsh) has the value of the last command that was run.**

Another look at Error Checking

- **You can use alias to improve error checking:**

```
#pErr prints a message if an error occurs.
alias pErr 'set ev=$status && echo Error: " $ev && exit $ev'

mv foo $ARCHIVE || pErr
```

Seeing if a variable is set

```
if ( $?ARCHIVE ) then
    echo \$ARCHIVE is not set!
endif
```

Here `\$` ensures the "`$`" is not used to dereference `ARCHIVE`.

You could also use '`$ARCHIVE ...`'

foreach

Foreach iterates on an array.

```
foreach f (/usr/bin/*)
    if ( -f $f && ! -l $f ) then
        echo $f
    endif
end

set arr=(a b c)
foreach v ($arr)
    echo $v
end
```

while

```
#handle commandline arguments (default is 10 with array
#syntax)
while ( $#argv )
    if ( -d ${argv[1]}) then
        echo ${argv[1]} is a directory!
    endif
    shift
end
# simulate a regular C for loop.
set ii=0
while ( $ii < 10 )
    echo $ii
    @ ii=($ii + 1)
end
```

String Operators

- **C-Shell has a group of operators that can act on strings.**
- **E.g. get the root and extension of a file.**

```
% set f="/u1/uaf/username/bath.nc"
% echo $f
/u1/uaf/username/bath.nc
% echo $f:r
/u1/uaf/username/bath
% echo $f:e
nc
```

- **True csh this does not work for environment variables (tcsh does).**

String Operators

- **Other operators**
- `:r` **(root, part of string before last dot)**
- `:e` **(extension, part of string after last dot)**
- `:h` **(part of the string before last "/")**
- `:t` **(part of string after last "/")**
- `:gr`**,** `:ge`**,** `:gh`**,** `:gt` **(perform operations above on an array of files** `g`**=global)**