Shell Programming

Why shell programming (aka scripting)?

- *Simplicity*: Often work at a higher level than compiled languages (easier access to files and directories)
- *Ease of development*: Can code a problem in a fraction of the time it would take to code in C,C++,Java
- *Portability*: Easy to make them portable with little or no modification
- *Disadvantage*: Less efficient than statically typed languages
- *Disadvantage*: Doesn't scale to bigger projects

Often used to:

- Automate system administration tasks
- Combine existing programs to accomplish a single task (e.g. black box)

Creating a New Command

We can create new shell commands by writing a *shell script*.

- **Example:** Define a `hello` command that will print "Hello World" to the console.

lab/shell-scripts/hello.sh

```
#!/bin/bash
# lab/shell-scripts/hello.sh

STR="Hello World"
echo $STR
```

- Now we can set the executable bit and run it like any other command.

```
  [amit@onyx]: chmod +x hello.sh
  [amit@onyx]: hello.sh
 Hello world
```

Shell Script Basics

- Shell scripts typically have `.sh` extension, but not required.
- The first line of a script is called the *shabang* or *shebang*.
  - Defines the path to the command interpreter so the operating system knows how to execute the file
  - Has the form `#!/path/to/interp <flags>`
  - Bash script has the form `#!/bin/sh`
  - May be used with any interpreter (such as `python`), not just shell languages
- Comments also begin with `#`. They must occur after the *shabang*.

Running shell scripts

- The executable bit must be set on the file before executing.

```
chmod +x <script>
```

- It is usually a good idea (for security) to use the full path to execute a script; if it is in your current directory you can execute as

```
./<script>
```

- As it stands, `hello.sh` only works if it is in your current directory (assuming the current directory is in your `PATH`).
- To run a shell script from anywhere, move the script to your `~/bin/` directory. (This works because `~/bin/` is on the shell `PATH`).
- Run with `-x` to display the commands as they are executed.

```
bash -x <script>
```

Shell metacharacters

Character	Meaning
'...'	Take literally without interpreting contents
“...”	Take literally after processing $, \`...\` and \
\	Escape, for example \c takes character c literally
\`...\`	Run enclosed command and replace with output

Shell Variables

- Shell variables are created when assigned.
- Assignment statement has *strict syntax*.
    - No spaces around = sign.
    - Assigned value must be a single word (quoted if necessary).
- After a variable is defined, it may be referenced by prefixing the name with the $ symbol (e.g. `$var`).

```
STR="Hello World"
echo $STR
```

- Variables defined in the shell can be made available to scripts and programs by exporting them as *environment* variables.

```
export HOME=/home/amit
```

- Any script or program we run during this session will use the new value.

Shell Variables (2)

- Some shell variables are predefined when you log in.

Variable	Evaluates to...
`PATH`	the value of the `PATH` environment variable
`HOME`	the full path name of your home directory
`USER`	your user name
`PWD`	the current directory path
`$$`	the process-id of the shell script
`$?`	the return value of the last command

lab/shell-scripts/vars.sh

```
#!/bin/bash
# lab/shell-scripts/vars.sh
echo "HOME="$HOME
echo "USER="$USER
echo "PATH="$PATH
echo "PWD="$PWD
echo "\$\$"=$$
```

Command Line Arguments

- The shell also provides built-in variables for accessing script parameters.

Variable	**Evaluates to...**
$0	Name of the current script
$*x*	The value of the *x*'th command line argument
$*	Space-separated list of all command line arguments
$#	Number of command line arguments

Using Command Line Arguments

lab/shell-scripts/lc.sh

```
#!/bin/bash
# lab/shell-scripts/lc.sh

echo Executing $0
echo $# files
wc -l $*
```

- Note that $0 gives the pathname of the script as it was invoked.

```
[marissa@onyx]: ./lc.sh *
[marissa@onyx]:
Executing lc.sh...
    26 files
    24 backup1.sh
    33 backup2.sh
    ...
```

Program output as an argument

What if we wanted to use the output of the `ls` command as input to our `lc.sh` script?

- We can expand program/command output using `$(cmd)` (*preferred syntax*) or `` `cmd` `` (*alternative syntax*).
- So, we can pass the result of `ls` to our script from the command line

```
[marissa@onyx]: ./lc.sh $(/bin/ls)
```

- Or use the the evaluation of a program as an argument to command in our script.

lab/shell-scripts/lc-pwd.sh

```
#!/bin/bash
# lab/shell-scripts/lc-pwd.sh

echo Executing $(basename $0)...
echo $(/bin/ls | wc -l) files
wc -l $(/bin/ls)
```

Test

- The `test` command may be used to evaluate expressions and check file properties.
- Alternative is to enclose expression in `[ ]`'s.

```
test EXPRESSION
or
[ EXPRESSION ]
```

- Returns the exit status of the given expression.
- For more information see `man test`.
- May be used with any of the test expressions on the following slides.
- The *extended test command* `[[...]]` was introduced in Bash 2.02. It can simplify expression syntax for those familiar with Java and C. See online Bash documentation.

String Comparison Operators

Operator	True if...
str1 = *str2*	*str1* matches *str2*
str1 != *str2*	*str1* does not match *str2*
str1 < *str2*	*str1* is less than *str2*
str1 > *str2*	*str1* is greater than *str2*
-n *str1*	*str1* is not null (length >0)
-z *str1*	*str1* is null (length = 0)

```
test "$1" = "hello"
or
[ "$1" = "hello" ]

test -n "$1"
or
[ -n "$1" ]
```

Integer Conditionals

Test	Comparison
`-lt`	Less than
`-le`	Less than or equal
`-eq`	Equal
`-ge`	Greater than or equal
`-gt`	Greater than
`-ne`	Not equal

```
test $# -gt 0
or
[ $# -gt 0 ]

test $1 -eq 2
or
[ $1 -eq 2 ]
```

File Attributes

Operator	True if...
`-a` *file*	*file* exists
`-d` *file*	-d *file* exists and is a directory
`-e` *file*	*file* exists; same as -a
`-f` *file*	*file* exists and is a regular file (not a directory, ...)
`-r` *file*	You have read permission on *file*
`-s` *file*	*file* exists and is not empty
`-w` *file*	You have write permission on *file*
`-x` *file*	You have execute permission on *file*
`-N` *file*	*file* was modified since last read
`-O` *file*	You own *file*
`-G` *file*	file's group ID matches yours
file1 `-nt` *file2*	file1 is newer than file2
file1 `-ot` *file2*	file1 is older than file2

File Attributes: Example

lab/shell-scripts/file-info.sh

```
#!/bin/bash

if [ ! -e "$1" ]; then
    echo "file $1 does not exist."
    exit 1
fi

if [ -d "$1" ]; then
    echo -n "$1 is a directory that you may "
    if [ ! -x "$1" ]; then
        echo -n "not "
    fi
    echo "search."
elif [ -f "$1" ]; then
    echo "$1 is a regular file."
else
    echo "$1 is a special type of file."
fi

if [ -O "$1" ]; then
    echo 'you own the file.'
else
    echo 'you do not own the file.'
fi

if [ -r "$1" ]; then
    echo 'you have read permissions on the file.'
fi

if [ -w "$1" ]; then
    echo 'you have write permission on the file.'
fi

if [ -x "$1" -a ! -d "$1" ]; then
    echo 'you have execute permission on the file.'
fi
```

Compound Comparison

Operator	Meaning
`-a`	Logical `and`
	expr1 -a expr2 returns true if expr1 and expr2 are true.
`-o`	Logical `or`
	expr1 -o expr2 returns true if either expr1 or expr2 is true.
`!`	Logical `not`
	! expr1 returns the invert of expr1.

```
test $# -gt 0 -a "$2" = "hello"
or
[ $# -gt 0 -a "$2" = "hello"]

test $# -gt 0 -o "$2" = "hello"
or
[ $# -gt 0 -o "$2" = "hello"]
```

Arithmetic Expressions

- Normally variables in shell scripts are treated as strings.
- To use numerical variables, enclose expressions in square brackets.
- We can also use the `let` keyword to do an arithmetic operation.

Arithmetic (lab/shell-scripts/arithmetic.sh)

```
#!/bin/sh

sum=0
sum=$[sum + 1]
echo $sum

let sum++
echo $sum
```

Arithmetic Operators

Operator	**Meaning**
++	Increment by one (prefix and postfix)
--	Decrement by one (prefix and postfix)
+	Plus
-	Minus
*	Multiplication
/	Division (with truncation)
%	Remainder
»	Bit-shift left
«	Bit-shift right
&	Bitwise and
|	Bitwise or
∼	Bitwise not
!	Logical not
∧	Bitwise exclusive or
,	Sequential evaluation

Loops and conditional statements (1)

`if/else`:	execute a list of statements if a certain condition is/is not true
`for`:	execute a list of statements a fixed number of times
`while`:	execute a list of statements repeatedly while a certain condition holds true
`until`:	executes a list of statements repeatedly until a certain condition holds true
`case`:	execute one of several lists of statements depending on the value of a variable
`select`:	Allows the user to select one of a list of possibilities from a menu

Loops and conditional statements (2)

if/else (lab/shell-scripts/flow-control/fc1.sh)

```
#!/bin/bash
A="a"
B="b"
if [ "$A" \> "$B" ];then # > < must be escaped with \
    echo "$A > $B"
elif [ "$A" \< "$B" ]; then
    echo "$A < $B"
else
    echo "$A = $B"
fi
```

Loops and conditional statements (3)

for (lab/shell-scripts/flow-control/fc2.sh)

```
#!/bin/bash

for f in *.sh; do
    echo $f
done
```

for (lab/shell-scripts/flow-control/fc7.sh)

```
#!/bin/bash

for ((i=0; i<10; i++)); do
    echo $i
done
```

Loops and conditional statements (4)

`while` (lab/shell-scripts/flow-control/fc3.sh)

```
#!/bin/bash

i=0;
while [ $i -lt 10 ]; do
    let i++
    echo $i
done
```

`until` (lab/shell-scripts/flow-control/fc4.sh)

```
#!/bin/bash

i=10;
until [ $i -lt 1 ]; do
    let i--
    echo $i
done
```

Loops and conditional statements (5)

case (lab/shell-scripts/flow-control/fc5.sh)

```
#!/bin/bash

case "$1" in
        start)
            echo "in start..."
            ;;
        stop)
            echo "in stop..."
            ;;
        status)
            echo "in status..."
            ;;
        restart)
            echo "in restart"
            ;;
        *)
            echo $"Usage: $0 {start|stop|restart|status}"
            exit 1

esac
```

Loops and conditional statements (6)

select (lab/shell-scripts/flow-control/fc6.sh)

```
#!/bin/bash

printf "Select your favorite animal:\n"
select term in \
    'Dolphin' \
    'Panda' \
    'Tiger' \
    'Bronco'
do
    case $REPLY in         #REPLY is built in variable
        1 ) NAME="Flipper the Dolphin";;
        2 ) NAME="Happy Panda";;
        3 ) NAME="Tony the Tiger";;
        4 ) NAME="Buster Bronco";;
        * ) printf "invalid." ;;
    esac
    if [[ -n $term ]]; then
        printf "$NAME is your favorite animal!\n"
        break
    fi
done
```

Interactive Programs

- If a program reads from its standard input, we can use the "here document" concept in shell scripts.
- Suppose we have a program `p1` that reads two integers followed by a string.
- We can orchestrate this in our script as follows:

```
#!/bin/bash

p1 <<END
12 22
string1
END
```

- `END` is an arbitrary token denoting the end of the input stream to the program `p1`.

Functions (1)

- Generally, shell functions are defined in a file and sourced into the environment as follows:

```
$ .  file
```

or

```
$ source file
```

- They can also be defined from the command line. The syntax is simple:

```
name () {
  commands;
}
```

- Parameters can be passed, and are referred to as $1, $2, and so on.
- $0 holds the function name.
- $# refers to the number of parameters passed.
- $* expands to all parameters passed.

Functions (2)

- Since functions affect the current environment, you can do things like this:

```
tmp () {
  cd /tmp
}
```

- This will `cd` to the `/tmp` directory.
- You can define similar functions to `cd` to directories and save yourself some typing. This can't be done in a shell script, since the shell script is executed in a subshell.
- This means that it will `cd` to the directory, but when the subshell exits, you will be right where you started.

Functions (3)

- Functions can be recursive.

Recursive Function (lab/shell-scripts/recursive.sh)

```
#!/bin/bash

holeinmybucket() {
    let depth++
    echo "depth = " $depth
    holeinmybucket
}

holeinmybucket
```

- What happens if you run the above function?

Function Arguments (1)

- Here is a function that uses arguments:

```
add () {
  echo $[$1 + $2];
}
```

- To use the function:

```
$ add 2 2
4
$
```

Function Arguments (2)

- The following example shows that the notion of arguments is context-dependent inside a function.

Function Arguments (lab/shell-scripts/functionArgs.sh)

```
#!/bin/bash

echoargs ()
{
    echo '=== function args'
    for f in $*
    do
        echo $f
    done
}

echo --- before function is called
for f in $*
do
    echo $f
done

echoargs a b c

echo --- after function returns
for f in $*
do
    echo $f
done
```

Function Arguments (3)

- The output of the previous example.

Function Arguments (lab/shell-scripts/functionArgs.sh)

```
[marissa@onyx shell-scripts]\$ ./functionArgs.sh 1 2 3
--- before function is called
1
2
3
=== function args
a
b
c
--- after function returns
1
2
3
```

Example: Changing file extensions in one fell swoop

Suppose we have hundreds of files in a directory with the extension .cpp and we need to change all these files to have an extension .cc instead. The following script mvall does this if used as follows.

```
mvall cpp cc
```

mvall.sh (lab/shell-scripts/mvall/mvall.sh)

```
#!/bin/bash

if [ $# -le 1 ]
then
    echo "Usage " $(basename $0) " <oldext> <newext>"
    exit 1
fi

#case $# in
#0|1)
#   echo "Usage " $(basename $0) " <oldext> <newext>"
#   exit 1
#esac

oldext=$1
newext=$2

for f in *.$oldext
do
  echo $f
  base=$(basename $f .$oldext) # e.g. f1.cpp => f1
  mv $f $base.$newext
done
```

- The for loop selects all files with the given extension.
- The basename command is used to extract the name of each file without the extension.
- Finally the mv command is used to change the name of the file to have the new extension.

Example: Replacing a word in all files in a directory (1)

A directory has many files. In each of these files we want to replace all occurrences of a string with another string. We only want to do this for regular files.

```
changeword /bin/sh /bin/bash
```

changeword.sh (lab/shell-scripts/changeword/changeword.sh)

```
#!/bin/sh
# sed/changeword

prog=`basename $0`
case $# in
0|1) echo 'Usage:' $prog '<old string> <new string>'; exit 1;;
esac

old=$1
new=$2
for f in *
do
        if [ "$f" != "$prog" ]
        then
            if [ -f "$f" ]
            then
                sed "s/$old/$new/g" $f > $f.new
                mv $f $f.orig
                mv $f.new $f
                echo $f done
            fi
        fi
done


# a simpler version

#for f in *
#do
#        if [ "$f" != "$prog" ]; then
#            if [ -f "$f" ]; then
#                sed -i.orig "s/$old/$new/g" $f
#                echo $f done
#            fi
#        fi
#done
```

Example: Replacing a word in all files in a directory (2)

- The `for` loop selects all files in the current directory.
- The first `if` statement makes sure that we do not select the script itself!
- The second `if` tests to check that the selected file is a regular file.
- Finally we use `sed` to do the global search and replace in each selected file.
- The script saves a copy of each original file (in case of a problem).

Example: Counting files greater than a certain size (1)

For the current directory we want to count how many files exceed a given size.

- For example, to count how many files are greater than or equal to 100K in the current folder.

```
countsize.sh .  100
```

countsize.sh (lab/shell-scripts/countsize.sh)

```
#!/bin/sh

prog=$(basename $0)
case $# in
0|1) echo "Usage: $prog <folder> <size in K>"; exit 1;;
esac

folder=$1
limit=$2
limitinbytes=$[limit*1024]
count=0

cd $folder
for f in *
do
    if [ -f $f ]
    then
        size=$(ls -l $f| awk '{print $5}')
    if  [ $size -ge $limitinbytes ]
        then
            count=$[count+1]
            echo $f $size
        fi
    fi
done
echo $count "files bigger than " $limit"K" " found in folder $folder"
```

Example: Counting files greater than a certain size (2)

- First, we convert the given number of kilobytes to bytes.
- For each selected file, the first `if` checks if it is a regular file.
- Then we use the command `ls -l $f | awk 'print $5'`, which prints the size of the file in bytes.
- We pipe the output of the `ls` to `awk`, which is used to extract the first field (the size) and put this pipe combination in back-quotes to evaluate and store the result in the variable `size`.
- The `if` statement then tests if the size is greater than or equal to the limit. If it is, then we increment the `count` variable. (Note the use of the square brackets to perform arithmetic evaluation.

Example: Counting number of lines of code recursively

The following script counts the total number of lines of code of all `.c` files starting in the current directory and continuing in the subdirectories recursively.

- For example, if we wanted to count the number of lines in all `.c` files in `lab/C-examples/intro` directory (*assuming countlines.sh is in your bin directory*).

```
$ cd lab/C-example/intro
$ countlines.sh
```

`countlines.sh` (lab/shell-scripts/countlines.sh)

```
#!/bin/bash

total=0
for currfile in $(find . -name "*.c" -print)
do
    total=$[total+($(wc -l $currfile| awk '{print $1}'))]
    echo -n 'total=' $total
    echo -e -n '\r'
done
echo 'total=' $total
```

- If you want to be able to count `.h`, `.cc` and `.java` files as well, modify the argument `-name "*.c"` to `-name "*.[c|h|cc|java]"`

Example: Backing up your files periodically

The following script periodically (every 15 minutes) backs up a given directory to a specified backup directory. You can run this script in the background while you work in the directory.

- An example use may be as shown below.

```
backup1.sh cs253 /tmp/cs253.backup &
```

backup1.sh (lab/shell-scripts/backup1.sh)

```
#!/bin/sh
# recursive/backup1.sh

prog=$(basename $0)
case $# in
0|1) echo 'Usage:' $prog '<original dir> <backup dir>'; exit 1;;
esac

orig=$1
backup=$2
interval=900 #backup every 15 minutes

while true
do
    if test -d $backup
    then
        /bin/rm -fr $backup
    fi
    echo "Creating the directory copy at" $(date)
    /bin/cp -pr $orig $backup
    sleep $interval
done
```

Example: Backing up your files with minimal disk space (1)

- A simple backup script that creates a copy of a given directory by using *hard links* instead of making copies of files.
- This results in substantial savings in disk space. Since the backup file has hard links, as you change your files in the working directory, the hard links always have the same content.
- If you accidentally removed some files, you can get them from the backup directories since the system does not remove the contents of a file until all hard links to it are gone.
- Note that hard links cannot span across filesystems.

Example: Backing up your files with minimal disk space (2)

backup2.sh (lab/shell-scripts/backup2.sh)

```
#!/bin/sh
# backup2.sh

prog=$(basename $0)
case $# in
0|1) echo 'Usage:' $prog '<original dir> <backup dir>'; exit 1;;
esac

orig=$1
backup=$2
if test -d $backup
then
    echo "Backup directory $backup already exists!"
    echo -n "Do you want to remove the backup directory $backup? (y/n)"
    read answer
    if test "$answer" = "y"
    then
        /bin/rm -fr $backup
    else
        exit 1
    fi
fi

mkdir  $backup
echo "Creating the directory tree"
find $orig -type d -exec mkdir $backup/"{}" \;

#make hard links to all regular files
echo "Creating links to the files"
find $orig -type f -exec ln {} $backup/"{}" \;

echo "done!"
```

Example: Watching if a user logs in/out (1)

- The following script watches if a certain user logs in or out of the system.
- The following example usage which will watch if `amit` logs in or out every 10 seconds.

```
watchuser amit 10
```

watchuser.sh (lab/shell-scripts/watchuser.sh)

```
#!/bin/bash

case $# in
0) echo 'Usage: ' $prog '<username> <check interval(secs)>'; exit 1;;
esac

name=$1
if test "$2" = ""; then
    interval=60
else
    interval=$2
fi

who | awk '{print $1}' | grep $name >& /dev/null

if test "$?" = "0"; then
    loggedin=true
    echo $name is logged in
else
    loggedin=false
    echo $name not logged in
fi

while true
do
    who | awk '{print $1}' | grep $name >& /dev/null
    if test "$?" = "0"; then
        if test "$loggedin" = "false"; then
        loggedin=true
            echo $name is logged in
        fi
    else
        if test "$loggedin" = "true"; then
            loggedin=false
            echo $name not logged in
        fi
    fi
    sleep $interval
done
```

Example: Watching if a user logs in/out (2)

- Here is another version, written using functions:

watchuser-with-fns.sh (lab/shell-scripts/watchuser-with-fns.sh)

```
#!/bin/bash

check_usage() {
    case $# in
    0) echo 'Usage: ' $prog '<username> <check interval(secs)>'; exit 1;;
    esac
}

check_user() {
    who | awk '{print $1}' | grep $name >& /dev/null
    if test "$?" = "0"; then
        if test "$loggedin" = "false"; then
            loggedin=true
            echo $name is logged in
        fi
    else
        if test "$loggedin" = "true"; then
            loggedin=false
            echo $name not logged in
        fi
    fi
}

check_usage $*

name=$1
if test "$2" = ""; then
    interval=60
else
    interval=$2
fi

loggedin=false
check_user $name

while true
do
    check_user $name
    sleep $interval
done
```

References

- *Local Linux Guide*. http://cs.boisestate.edu/~amit/teaching/handouts/cs-linux.pdf
- *Bash Scripts* by James Kress. A presentation on shell scripting by a past CS student.
- *Advanced Bash-Scripting Guide*. http://www.tldp.org/LDP/abs/html/
- *Shell Scripting* by Arnold Robbins and Nelsonn H.F. Beebe, O'Reilly.