Operating systems and concurrency - B02

Michael Brockway, David Kendall, Alun Moon

Northumbria University

Outline

- The Shell
- Shell scripts
- Command sequence
- Command line arguments
- Shell variables
- Conditions and if statements
- Loops - while, for
- Integer arithmetic

The Shell

- The shell is a *command interpreter*
- It acts as an interface between the user and the operating system kernel
- Usually,
  - The user enters lines at the terminal
  - The terminal passes each line to the shell which tries to *interpret* the line as a command
  - The shell makes calls to the operating system to execute the command; this may generate some output that is sent to the display
  - The shell returns a result
- The shell can perform sophisticated parameter expansion and command substitution and I/O redirection
- There is an IEEE/Posix Standard, IEEE Std 1003.1-2008, 2016 Edition that defines the standard behaviour of a shell
- The shell command language is defined as part of the POSIX standard

The shell

- Operating systems such as Linux, Mac OS X, FreeBSD, etc. allow the user to choose one of a number of different shells. There are two distinct types:
- Bourne-compatible shells
  - `sh`
  - `bash`
  - `ksh`
  - `zsh`
- C-shell-compatible shells
  - `csh`
  - `tcsh`
- The IEEE Posix standard defines the capabilities that a compliant shell should provide
- Even Microsoft are looking to provide a standard shell for Windows
- We focus on `bash` (Bourne Again SHell) which is able to be entirely POSIX-compliant
- C-shell syntax is regarded as being unsuitable for complex tasks

Shell scripts

- A *shell script* is an executable text file that contains shell commands
- This allow us to build new commands from already existing commands
- The shell command language has many features. We focus here on the basics. The references provided guidance for advanced features
- At its simplest, a script is just a sequence of commands...

```
#!/bin/bash
mkdir src
mv *.c* src
echo Source files copied to src folder
mkdir obj
mv *.o obj
echo Object files copied to obj folder
```

Shell scripts

Ignore the first line, `#!/bin/bash` for the moment.

The 6 remaining lines:

1. make a subfolder called `src`
2. moves all files with names ending “.c. . . " there (eg C or C++ source files)
3. echo a message on the console
4. make a subfolder called `obj`
5. moves all files with names ending “.o" there
6. echo another message on the console

This script, saved in a file `tidy`, is *run* at a Unix command prompt:

```
$ ./tidy
```

. . . and it does those 6 operations in order. This might be a useful script for tidying up a programming project working folder.

Make the script executable

Any sequence of Unix commands may be saved in a file and run like this.

The file's *execute* permission must be set –

- `-rwxrwxr-x 1 cgdk2 cgdk2 125 Jan 29 15:36 tidy`

You can switch on execute permission with the `chmod` command -

```
chmod a+x tidy
```

What about the first line, `#!/bin/bash`?

- This "shebang" line specifies which of several *shells* should provide the *command interpreter* to run the script: in this case *bash*
- If you don't specify which shell to use, the default shell will be used. This may, or may not, be what you intend
- Make this the first line of *all* your Unix scripts.

Command line arguments

- You can make your scripts more flexible by using command line arguments
- Here's a version of the `tidy` script that allows you to specify which directory should be tidied

```
cd $1
mkdir src
mv *.c* src
echo "Moved C and C++ files to src"
mkdir obj
mv *.o obj
echo "Moved object files to obj"

# Call this script tidy2 and execute it like this
# $ ./tidy2 progdir
# Notice the use of the positional argument $1
```

Command line arguments

- Each word on the command line, after the command, can be referred to in the script using the positional names, `$1, $2, $3, ...`
- You can refer to the argument list using `$*`
- The command name itself can be referred to using `$0`
- The number of arguments can be referred to using `$#`

```
echo "This $0 command has $# arguments"
echo "They are $*"
echo "The first argument is $1"
echo "The second argument is $2"
echo "The third argument is $3"

# Call it args and execute it like this
# $ ./args Fun with scripts
```

Shell variables

```
#!/bin/bash
# Example with command line arguments
# and shell variables
tidyDir=$1
srcDir=$2
objDir=$3

echo "Tidying directory $tidyDir ..."
mkdir $srcDir
mv $tidyDir/*.c* $srcDir
echo "Moved C and C++ files to $srcDir"
mkdir $objDir
mv $tidyDir/*.o $objDir
echo "Moved object files to $objDir"
# Call this script tidy3 and execute it like this
# $ ./tidy3 prog src obj
```

Shell variables

- The variables in the `tidy3` script are `tidyDir, srcDir` and `objDir`
- Variables do not need to be declared
- Assign a value to them using `=`
- Notice there are no spaces before or after `=`
- Prefix the variable name with a `$` to refer to its value

Conditional expressions (tests) and if statements

```
#!/bin/bash
if [ $# -ne 3 ]; then
  echo "Your command: $0 $*"
  echo "Usage: tidy3 <tidy dir> <src dir> <obj dir>"
  exit 1
else
  tidyDir=$1
  srcDir=$2
  objDir=$3
fi

echo "Tidying directory $tidyDir ..."
mkdir $srcDir
mv $tidyDir/*.c* $srcDir
echo "Moved C and C++ files to $srcDir"
mkdir $objDir
mv $tidyDir/*.o $objDir
echo "Moved object files to $objDir"
```

Conditional expression (tests)

- A test is written in between square brackets `[ ... ]`
- Notice the space after `[` and before `]`
- This previous example shows a test of *integers* : relations
  `-lt -le -eq -ge -gt`
- Other useful tests

```
[ -e FILE ] # True if FILE exists
[ -d FILE ] # True if FILE exists and is a directory
[ -f FILE ] # True if FILE exists and is a regular fil
[ -r FILE ] # True if FILE exists and is readable
[ -w FILE ] # True if FILE exists and is writable
[ -x FILE ] # True if FILE exists and is executable
```

- Other tests are available (see Bash Guide for Beginners)

More tests and ifs

```
#!/bin/bash

if [ $# -ne 3 ]; then
  echo "Your command: $0 $*"
  echo "Usage: tidy4 <tidy dir> <src dir> <obj dir>"
  exit 1
else
  tidyDir="$1"
  srcDir="$2"
  objDir="$3"
fi

if [ -d "$tidyDir" ]; then
  echo "Tidying directory $1 ..."
else
  echo "$tidyDir does not exist or is not a directory"
  exit 1
fi
```

```
if [ ! -d "$srcDir" ]; then
  if [ -f "$srcDir" ]; then
    echo "$srcDir exists and is not a directory"
    exit 1
  else
    mkdir "$srcDir"
  fi
fi
CFILES=$tidyDir/*.c*
if stat -t $CFILES >/dev/null 2>&1; then
  mv $CFILES $srcDir;
  echo "Moved C and C++ files to $srcDir"
fi
```

```
if [ ! -d "$objDir" ]; then
  if [ -f "$objDir" ]; then
    echo "$objDir exists and is not a directory"
    exit 1
  else
    mkdir "$objDir"
  fi
fi
OBJFILES=$tidyDir/*.o
if stat -t $OBJFILES >/dev/null 2>&1; then
  mv $OBJFILES $objDir
  echo "Moved object files to $objDir"
fi
echo "... done"
```

Loops - the while loop

```
#!/bin/bash
read -p "Type your name, please -> " nm
echo "Hello $nm."
echo "What is the meaning of life, the universe, \
and everything?"
read -p "Please type the answer -> " answer
while  [ "$answer" != "42" ]
do
  read -p "No, try again! -> " answer
done
echo That\'s right
```

A much cleaner structured repetition construct. Note

- use of `do` and `done` to mark the beginning and end of the repeated part
- the test `[ ]` used in the same way as in conditionals.

A loop helper - shifting positional parameters

```
#!/bin/bash
# echos all run-time parameters by repeatedly shifting
while [ "$1" != "" ]
do
  echo $1
  shift
done
```

If this script is saved in file `shiftEx`, run it with some command-line arguments:

```
$ shiftEx one two buckle my shoe
```

You should see the command-line arguments `one`, `two`, `buckle`, `my`, `shoe` displayed one below another.

In this script, `shift` copies all the command-line parameters down one place: \$2 $\rightarrow$ \$1, \$3 $\rightarrow$ \$2, \$4 $\rightarrow$ \$3, etc. This is repeated by the script until all the arguments have been shifted down to position \$1 and displayed by `echo`.

Loops - the `for` loop

```
#!/bin/bash
# Given [$1] a folder to look in, and [$2]  a list of files,
# display details of each file and offer choice to keep/delete it

for f in $2
do
  procFiles1 "$1/$f"
done
```

$2 is a *list*: a string. The `for` command will break the string into words and execute the body of the scrip on each word.

This script, `procFiles0`, might be called like this:

```
$ procFiles0 workFolder "file1 file2 file3 file4"
```

It will iterate through the list of files (they presumably exist in the `workFolder`) and run “subroutine" script `procFiles1` on each one – ie, on `workFolder/file1` then `workFolder/file2` then `workFolder/file3` then `workFolder/file4`

Loops - the `for` loop

What is `procFiles1`? Here it is -

```
#!/bin/bash
ls -l "$1"

read -p "Delete(y/N)?" yn
if [ "$yn" == "y" ]
then
  rm "$1"
  echo $1 deleted
else
  echo $1 skipped
fi
```

It lists the details (`ls -l`) of the file given in the argument $1 and offers to delete it or leave (skip) it.

Note that this script is begin *called* repeatedly by `procFiles0`, although it can also be run directly.

Loops - the `for` loop

Instead of `procFiles0`, we could use `procFiles`:

```
#!/bin/bash
# Given [$1] a folder to look in display details of all
# files and for each file, offer choice of keeping it

for f in $1/*
do
  procFiles1 "$f"
done
```

This script uses “globbing" to make a list for the `for` command. The expression `$1/*` expands to a list of all files in directory `$1`. Another variation would be

```
for f in $(cat $2)
do
  procFiles1 "$1/$f"
done
```

Integer arithmetic

```
#!/bin/bash

total=0
while [ "$1" != "" ]
do
  total=$(($total+$1))
  shift
done
echo the answer is $total
```

Note the syntax for an arithmetic assignment:
`total=$(($total+$1))` with the double parentheses and the extra “$" sign.

We could make a script to multiply all the numbers entered on the command line – set `total` initially to 1 rather than 0 and in the loop, multiply: `total=$(($total*$1))`

References

- Shotts. W., *The Linux Command Line*, No Starch Press, 2016
- Kochan, S. and Wood, P., *Shell Programming in Unix, Linux and OS X: The Fourth Edition of Unix Shell Programming (Developer's Library)*, Addison Wesley, 2016

Get on-line help in Linux using `man` or `info`