Anthony Ventresque
anthony.ventresque@ucd.ie

Operating Systems
COMP30640

Bash

School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

Outline

- Bash Language
- Variables
- Simple Control Flows
- Parameters of Commands
- Nested Commands
- Important Variables
- Exit Code
- Alias
- Bash Configuration File
- Filtering File Names
- File system

Bash – Bourne Again Shell

- For us ***bash*** is ***the shell*** we've been using and we will use
- There are multiple shells: bash, tcsh, zsh, ksh, cmd.exe etc.

- Command processor (executes commands)
- Also a programming language
  - typical control flows
    - if, while, for, etc.
  - variables
- Efficient access to some of the kernel components: communication (e.g., pipes), file system etc.

A Bash “Text”

- A bash “text” or program is composed of bash words

- A bash word is composed of
  - characters separated by whitespace character: space, tab or newline
  - E.g., `Hello=42!*` is a single word
  - Exceptions:
    - `;` `&` `&&` `|` `||` `(` `)` `` ` `` do not require whitespaces
    - any string between “ ” or ’ ’ is considered a single word

A Bash Text

Here we have 5 words

"In Bash this is a unique word"

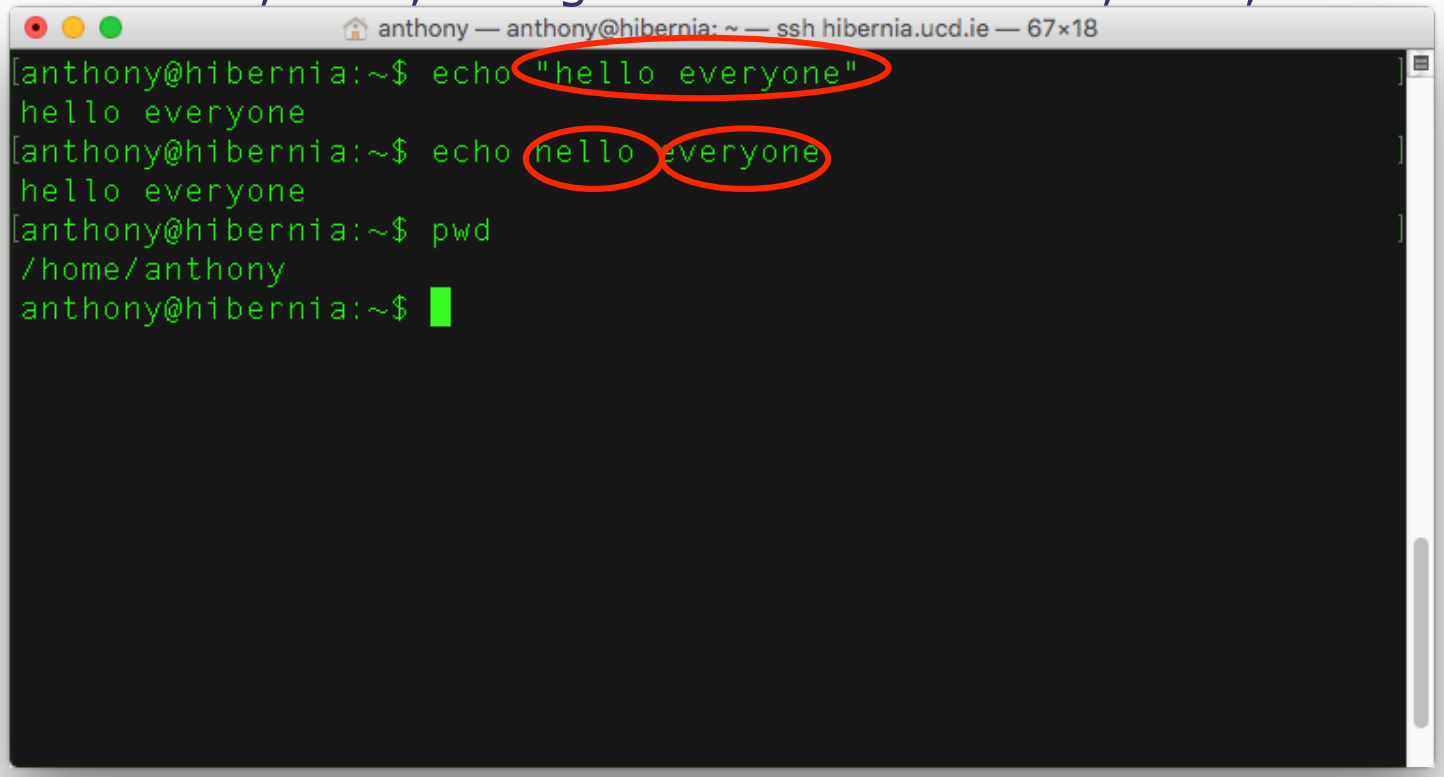
Here, three, words

We|have;NINE&&words&here

Remember the special characters which do not need whitespaces

Calling a Bash Command

- var1=value1 var2=val2 ***cmd*** arg1 arg2
  - only cmd is mandatory
  - everything else optional
  - runs cmd with parameters arg1, arg2, etc. and variables var1, var2, etc. given the values val1, val2, etc.



Special Characters

- Special characters
  - `\ ' ` " > < $ # * ~ ? ;( ) { }`
  - `'` is a single quote while `` ` `` is a back tick

- To prevent interpreting special characters
  - `\` disables interpreting the next special character
  - `'string'` disables interpreting the string
  - `"string"` the only characters that are considered special characters in the string are `$ \ ` `

Example

create 3 files

create 1 file

tests — -bash — 64×17

```
anthonys-MacBook-Air:tests anthony$ touch a b c
anthonys-MacBook-Air:tests anthony$ ls -l
total 0
-rw-r--r--  1 anthony  staff  0 29 Sep 08:30 a
-rw-r--r--  1 anthony  staff  0 29 Sep 08:30 b
-rw-r--r--  1 anthony  staff  0 29 Sep 08:30 c
anthonys-MacBook-Air:tests anthony$ rm *
anthonys-MacBook-Air:tests anthony$ touch "a b c"
anthonys-MacBook-Air:tests anthony$ ls -l
total 0
-rw-r--r--  1 anthony  staff  0 29 Sep 08:31 a b c
anthonys-MacBook-Air:tests anthony$ rm *
anthonys-MacBook-Air:tests anthony$ touch a\ b\ c
anthonys-MacBook-Air:tests anthony$ ls -l
total 0
-rw-r--r--  1 anthony  staff  0 29 Sep 08:31 a b c
anthonys-MacBook-Air:tests anthony$
```

Bash Script

- A script bash is a bash “text” in a text file
  - interpreted by bash when run by the user
  - can be modified in a text editor
  - a bash program needs to be made executable:

    chmod u+x my_script.sh
  - the .sh extension is just a convention
  - to execute the script:

    ./my_script.sh [arg1 arg2 …]

Structure of a Bash Script

- First line always contains the following:

  #!/bin/bash

  – this gives the kernel the path to the command processor

```
#!/bin/bash
instructions
instructions
[exit code]
```

Bash Variables

- Assign a variable using =: `my_var = value`
- $ is used to read a variable: `$my_var`
- interactive read: `read var1 var2 …`
  - reads some input given by the user (until new line character)
  - first word in var1
  - etc.

Bash Variable - example

anthony — anthony@hibernia: ~ — ssh hibernia.ucd.ie — 67×18

```
anthony@hibernia:~$ a=42
anthony@hibernia:~$ echo $a
42
anthony@hibernia:~$ s='hello world!!'
anthony@hibernia:~$ echo $s
hello world!!
anthony@hibernia:~$ read x
this is my text
anthony@hibernia:~$ echo $x
this is my text
anthony@hibernia:~$ read x y
this is my text
anthony@hibernia:~$ echo $x
this
anthony@hibernia:~$ echo $y
is my text
anthony@hibernia:~$
```

Bash = Imperative Programming

- Bash is an imperative programming language
  - sequence of instructions on different lines
  - or separated by ;

```
instruction1
instruction2
instruction3; instruction4
instruction5
```

Conditional Expression

```
if cond; then
   instructions
elif con; then
   instructions
else
   instructions
fi
```

- Each cond is a logical expression (true/false)

Logical Expressions

whitespaces are important!

- Logical expressions on numerical values
  - `[ n1 -eq n2 ]`: true if n1 is equal to n2
  - `[ n1 -ne n2 ]`: true if n1 is different from n2
  - `[ n1 -gt n2 ]`: true if n1 is greter than n2
  - `[ n1 -ge n2 ]`: true if n1 is greater than or equal to n2
  - `[ n1 -lt n2 ]`: true if n1 is less than n2
  - `[ n1 -le n2 ]`: true if n1 is less than or equal to n2
- Logical expressions on string
  - `[ word1 = word2 ]`: true if word1 equals word2
  - `[ word1 != word2 ]`: true if is word1 different than word2
  - `[ -z word ]`: true if word is an empty word
  - `[ -n word ]`: true if word is not an empty word

[cond]

`[ cond ]` is an alias for the command `test cond`

```
if [ $x -eq 42 ]; then
 echo coucou
fi
```

is equivalent to

```
if test $x -eq 42; then
 echo coucou
fi
```

Conditional Expression - example

```
#!/bin/bash
x=1
y=2
if [ $x -eq $y ]; then
 echo "$x = $y"
elif [ $x -ge $y ]; then
 echo "$x > $y"
else
 echo "$x < $y"
fi
```

Switch Expression

```
case word in
template1)
    instructions;;
template2)
    instructions;;
*)
    instructions;;
esac
```

- if word equals template1 then…
- else if word equals template2 then...
- default case...

Switch Expression - example

```
#!/bin/bash
res="en"
case $res in
"en")
    echo "Hello";;
"it")
    echo "Ciao";;
*)
    echo "dia dhuit";;
esac
```

While Loop

```
while cond; do
   instructions
done
```

- While condition cond is true do the instructions
- keyword break to exit the loop in case you need to…

While Loop - example

```
#!/bin/bash
x=10
while [ $x -ge 0 ]; do
    read x
    echo $x
done
```

For Loop

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

- For every element in the list
  - var gets assigned the next element
  - instructions are processed

For Loop - example

```
#!/bin/bash
for var in 1 2 3 4; do
   echo $var
done
```

Parameters of a Command

- ./my_script.sh arg1 arg2 arg3 …
- every word is stored in a variable

my_script.sh	arg1	arg2	arg3	arg4	…
"$0"	”$1"	”$2”	”$3”	”$4"	

- "$0": the command’s name
- "$1" … "$9": the parameters
- "$#": the number of parameters
- "$@": list of the parameters
- shift: shift the list of parameters

Example

```
#! /bin/bash
for i in $@; do
    echo $i
done
```

anthony — anthony@hibernia: ~ — ssh hibernia.ucd.ie — 67×12

```
anthony@hibernia:~$ ./test2.sh chip dale
chip
dale
anthony@hibernia:~$ ./test2.sh "end of" the section
end of
the
section
anthony@hibernia:~$
```

Nested Commands

- To gets the (text) output of a command cmd, use $(cmd)
  - different from $cmd (gives access to variable cmd not command)

anthony — anthony@hibernia: ~ — ssh hibernia.ucd.ie — 67×12

```
anthony@hibernia:~$ date
Wed Sep 28 23:21:22 IST 2016
anthony@hibernia:~$ echo "today is $(date)"
today is Wed Sep 28 23:21:52 IST 2016
anthony@hibernia:~$ echo "today is $date"
today is
anthony@hibernia:~$
```

Some Variables

- HOME: full, absolute, real, path
- PS1: primary prompt which is displayed before each command
- PS2: secondary prompt displayed when a command needs more input (e.g. a multi-line command).

anthony — anthony@hibernia: ~ — ssh hibernia.ucd.ie — 67×11

```
anthony@hibernia:~$ if [ 0 == 0 ]; then echo 'yes'; fi
yes
anthony@hibernia:~$ if
> [ 0 -eq 0 ]; then echo 'yes'; fi
yes
anthony@hibernia:~$ PS2="++++"
anthony@hibernia:~$ if
++++[ 0 -eq 0 ]; then echo 'yes'; fi
yes
anthony@hibernia:~$ PS1="anthony>>"
anthony>>
```

PATH

- PATH is an environment variable
- PATH is composed of a set of path separated by :
  - PATH=/bin:/usr/bin

. corresponds to current directory

- Whenever bash tries to run a command cmd:
  - if cmd contains a / then bash runs the executable command ./cmd.sh, /bin/cmd.sh
  - or else bash tries to locate cmd in all the directories in PATH
  - see command which

- Can we add . in PATH
  - on the plus side: no need to use ./ anymore in front of commands
  - BUT possibility to create viruses/malwares by naming scripts like known commands and if an admin/root runs them…

Exit Code

- Every process returns some code: exit code
  - can be used to test what happened in a program
  - most of the time the exit code is explained in the man page
  - the exit code of the most recent command is stored in $?

anthony — anthony@hibernia: ~ — ssh hibernia.ucd.ie — 67×12

```
anthony@hibernia:~$ pwd
/home/anthony
anthony@hibernia:~$ echo $?
0
anthony@hibernia:~$ ls titi
ls: cannot access titi: No such file or directory
anthony@hibernia:~$ echo $?
2
anthony@hibernia:~$
```

Bash Alias

- Used to redefine/simplify commands
  - Creation: alias cmd='…'
  - Deletion: unalias cmd
  - Describe: alias cmd
  - List: alias

anthony — anthony@hibernia: ~ — ssh hibernia.ucd.ie — 67×12

```
anthony@hibernia:~$ alias ll='ls -l'
anthony@hibernia:~$ alias ll
alias ll='ls -l'
anthony@hibernia:~$ unalias ll
anthony@hibernia:~$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ls='ls --color=auto'
anthony@hibernia:~$
```

Configuration Files

- Executed at the start of bash
  - modifying them requires restarting bash OR using the command source
- Configuration files:
  - global /etc/profile
  - local (per user) ~/.bashrc
- Variables assignment, aliases, etc.