

Anthony Ventresque

Big Data Programming

anthony.ventresque@ucd.ie

COMP47470

CLI (Bash) for Big Data



School of Computer Science,
UCD

Scoil na Ríomheolaíochta,
UCD

What Does an Operating System Do?

- An OS is a ***program*** that acts as an ***intermediary*** between a user of a computer and the computer hardware
- Goals: Execute user programs, make the computing system easy to use, utilise hardware efficiently
- OS is:
 - ***Resource allocator***: decides between conflicting requests for efficient and fair resource use
 - ***Control program***: controls execution of programs to prevent errors and improper use of computer



User OS Interfaces

- Almost all operating systems have a user interface (UI).
 - ***Command-Line*** (CLI),
 - ***Graphics User Interface*** (GUI),
 - ***Batch*** (non-interactive, can be organised by job schedulers, workflow managers etc.)



CLI: Bash - Bourne-again shell

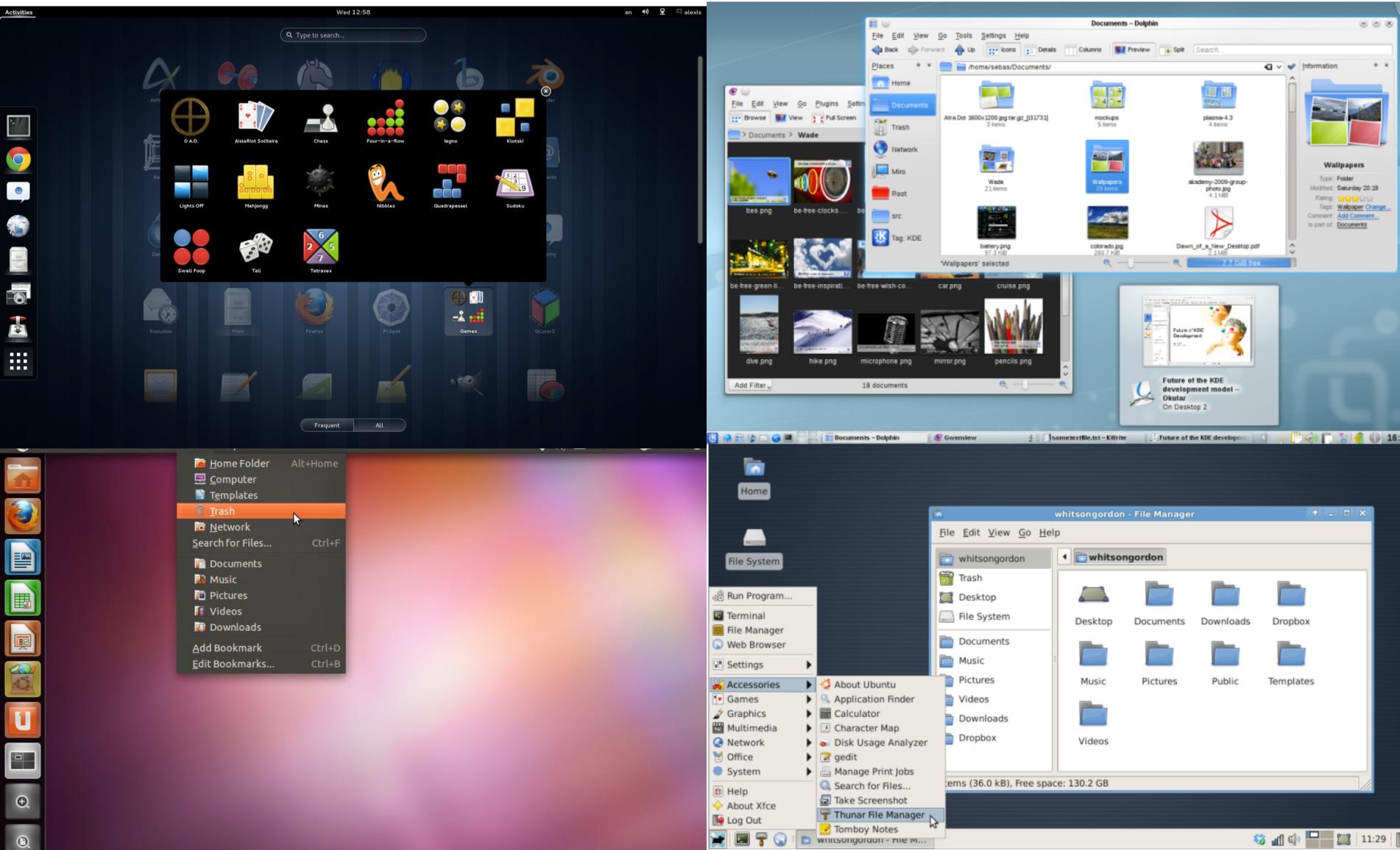
```
anthon...@hibernia:~$ w
21:29:27 up 94 days, 14:20, 1 user, load average: 0.01, 0.05, 0.05
USER    TTY      FROM          LOGIN@    IDLE   JCPU   PCPU WHAT
anthon... pts/7    89.100.150.232  21:23    0.00s  0.21s  0.00s w
[anthon...@hibernia:~$ iostat
Linux 3.2.0-40-generic (hibernia)        18/09/16      _x86_64_        (4 CPU)

avg-cpu: %user  %nice  %system %iowait  %steal  %idle
          2.33    0.00    0.09    0.03    0.00   97.55

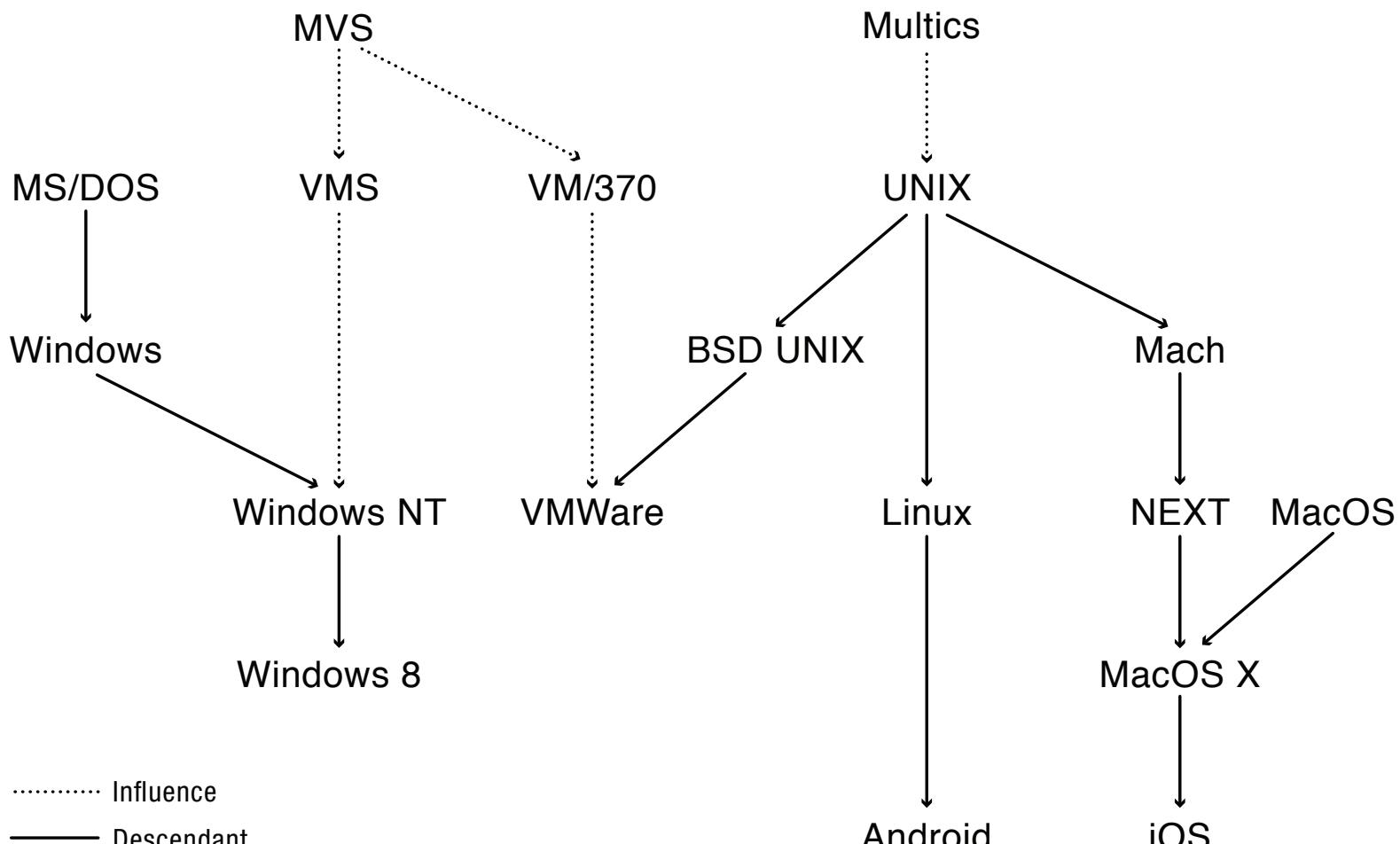
Device:      tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda       1.57     172.47      8.91 1409649758    72858524
sdb       1.56     172.45      8.93 1409454209    73002708
md0       1.23      0.92      7.66  7525125    62590088
dm-0       0.01      0.01      0.02    87340     144184

[anthon...@hibernia:~$ tracepath www.google.com
1: hibernia.ucd.ie                                0.084ms pmtu 1500
1: 193.1.132.2                                    136.114ms
1: 193.1.132.2                                    7.046ms
2: dc-er1-vlan13.ucd.ie                           0.666ms
3: te0-5-0-4-cr2-cwt.heanet.net                  2.184ms
4: heanet-ias-geant-gw.lon.uk.geant.net          10.884ms
5: ae0.mx1.ams.nl.geant.net                      25.400ms asymm 8
6: google.mx1.fra.de.geant.net                   25.401ms
7: no reply
8: no reply
9: no reply
10: no reply
11: no reply
12: no reply
```

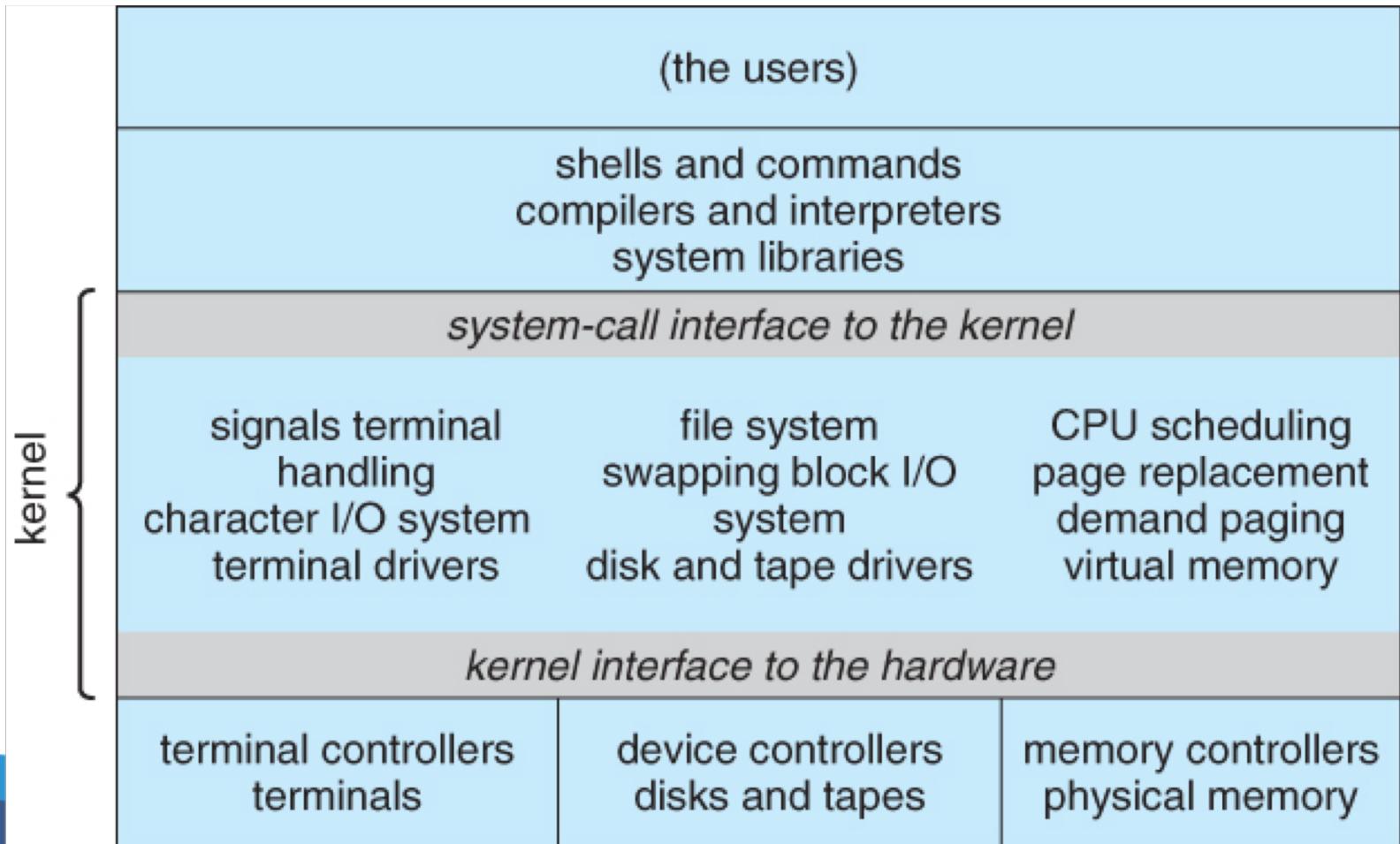
GUI: Gnome vs KDE vs Unity vs XFCE vs etc.



Desktop OS History



Unix System Structure



Bash – Bourne Again Shell

- For us **bash** is ***the shell***
- 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., Hello42!* 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

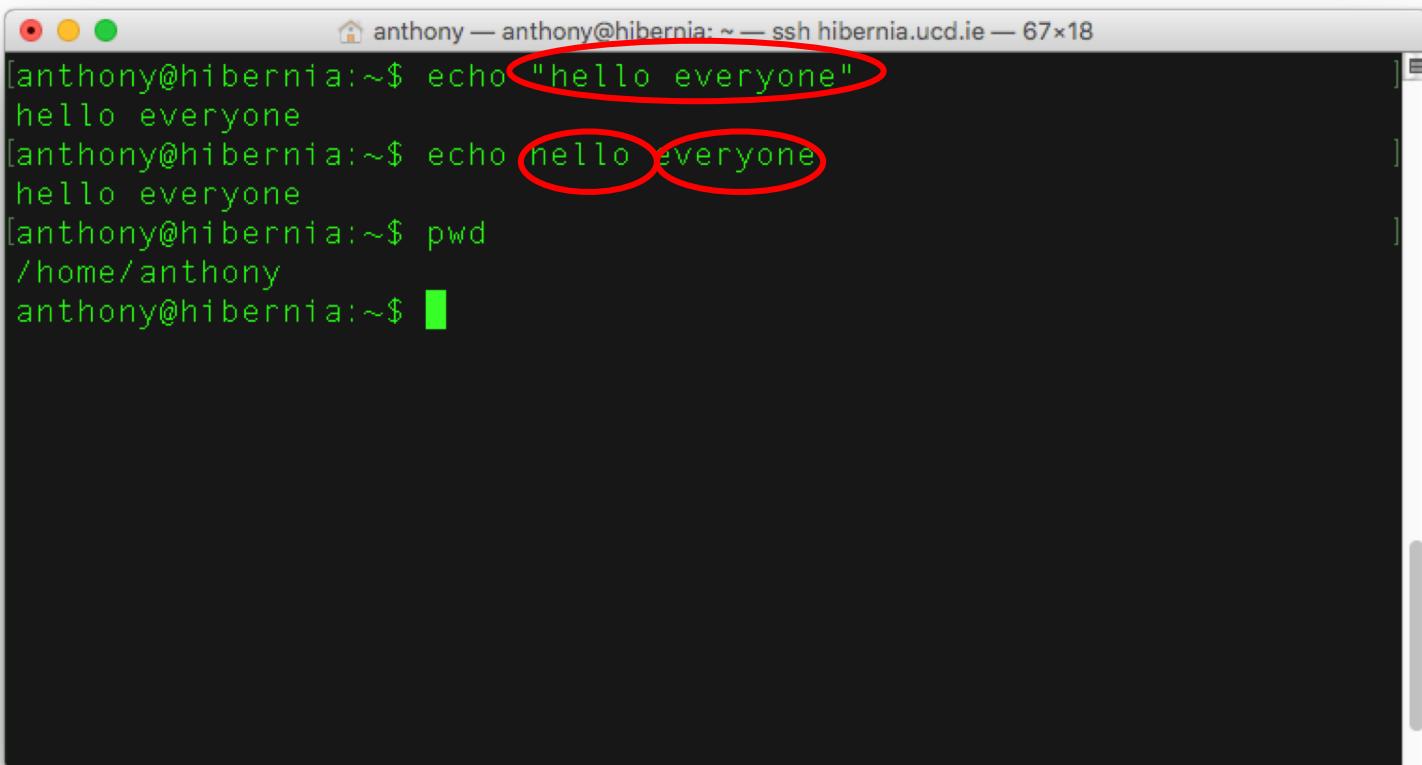
We|have;NINE&&words&here

Remember the special
characters which do not
need whitespaces



Calling a Bash Command

- ***cmd*** arg1 arg2
 - only cmd is mandatory
 - everything else optional
 - runs cmd with parameters arg1, arg2, etc.



The screenshot shows a terminal window with the following session:

```
anthony — anthony@hibernia: ~ — ssh hibernia.ucd.ie — 67x18
[anthony@hibernia:~$ echo "hello everyone"
hello everyone
[anthony@hibernia:~$ echo hello everyone
hello everyone
[anthony@hibernia:~$ pwd
/home/anthony
anthony@hibernia:~$ ]
```

Two lines of the command "echo hello everyone" are circled in red, highlighting the argument "hello everyone".



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 — 64x17
[anthony-MacBook-Air:tests anthony$ touch a b c
[anthony-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
[anthony-MacBook-Air:tests anthony$ rm *
[anthony-MacBook-Air:tests anthony$ touch "a b c"
[anthony-MacBook-Air:tests anthony$ ls -l
total 0
-rw-r--r-- 1 anthony staff 0 29 Sep 08:31 a b c
[anthony-MacBook-Air:tests anthony$ rm *
[anthony-MacBook-Air:tests anthony$ touch a\ b\ c
[anthony-MacBook-Air:tests anthony$ ls -l
total 0
-rw-r--r-- 1 anthony staff 0 29 Sep 08:31 a b c
anthony-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@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 greater 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 word1 is 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	...
<code>"\$0"</code>	<code>"\$1"</code>	<code>"\$2"</code>	<code>"\$3"</code>	<code>"\$4"</code>	

- `"$0"`: the command's name
- `"$1" ... "$9"`: the parameters
- `"$#"`: the number of parameters
- `"$@"`: list of the parameters
- `shift`: shift the list of parameters



Example

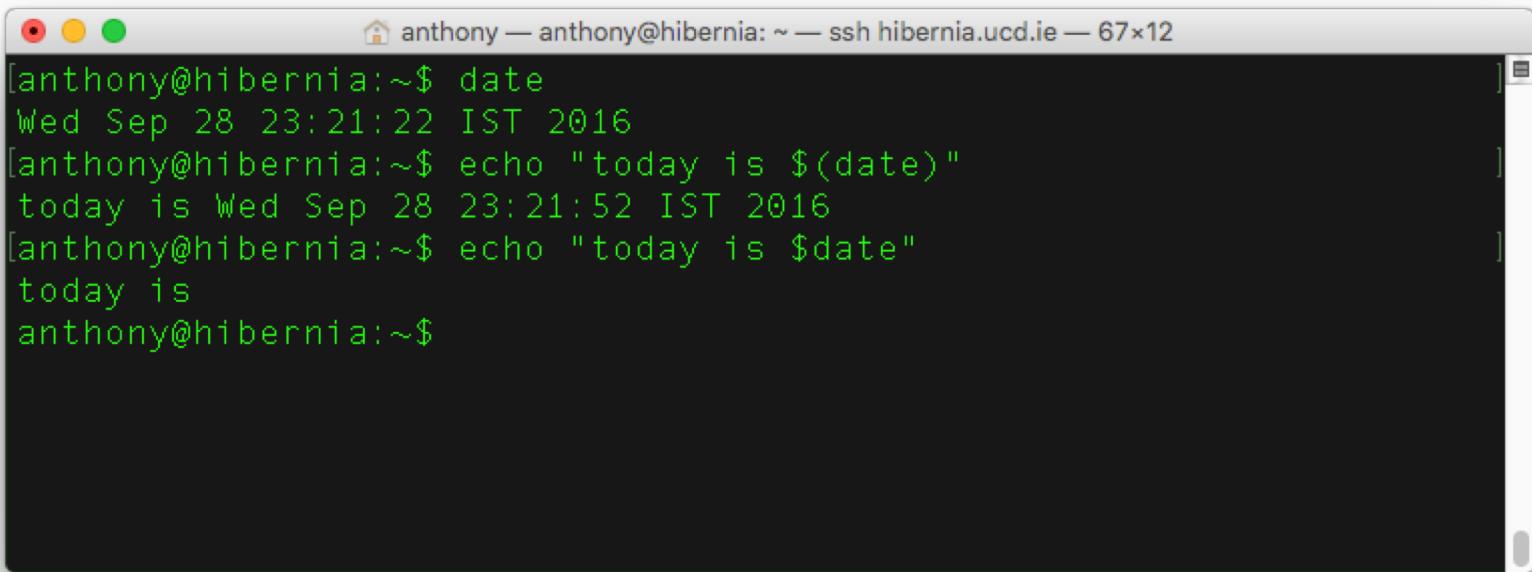
```
#!/bin/bash
for i in $@; do
    echo $i
done
```



```
anthony — anthony@hibernia: ~ — ssh hibernia.ucd.ie — 67x12
[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 get the (text) output of a command cmd, use \$(cmd)
 - different from \$cmd (gives access to variable cmd not command)



The screenshot shows a terminal window on a Mac OS X desktop. The title bar reads "anthon — anthon@hibernia: ~ — ssh hibernia.ucd.ie — 67x12". The terminal content is as follows:

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



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 — 67x11
[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 \$?



```
anthon — anthony@hibernia: ~ — ssh hibernia.ucd.ie — 67x12
[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 — 67x12
[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.



Program Input and Output

- Each program when executed has 3 files preset for it:
 - One input file, **STDIN** (usually the keyboard)
 - and 2 separate output files, **STDOUT** and **STDERR** (usually the console, terminal or shell window)
- (In Unix/Linux "everything is a file" - even the keyboard/screen)
- The program can of course completely ignore these and get its inputs, and set its outputs to anything it wants. But most of the standard Unix/Linux command line tools follow a convention in the use of these preset files.
 - If there is no alternative input file(s) specified, then the program takes its input from STDIN
 - It directs its normal output to STDOUT
 - It directs any error messages to STDERR



Program Input and Output



Redirection

- When executing programs, the SHELL can change what "file" is connected to STDIN, STDOUT and STDERR.
- Redirecting STDOUT
 - Use the '>' or '>>' symbols to change where the normal output goes. '>>' causes the output to be appended to any previous contents of the file.
 - \$> ls -l > ls_list
 - \$> ls -l /etc >> ls_list



Redirection

- When executing programs, the SHELL can change what "file" is connected to STDIN, STDOUT and STDERR.
- Redirecting STDIN
 - Use the '<' symbol to change where the normal input comes from
 - \$> wc -l < ls_list



Redirection

- When executing programs, the SHELL can change what "file" is connected to STDIN, STDOUT and STDERR.
- Redirecting STDERR
 - Use '2>' to change where the error output goes, 2 because STDERR is file 2. To append use '2>>'
 - \$> ls -l /home;brangelina 2> ls_list
 - \$> ls -l Z* 2> /dev/null
 - (note /dev/null is ~ a permanent bin)



Joining Programs Together: Pipes

- The shell can use a **PIPE** (see man pipe) to feed the output of one program into the input of another. This corresponds to joining STDOUT of one command with STDIN of a second command. Use the ' | ' symbol between the 2 commands.
 - \$> ls -l /etc | less

