Anthony Ventresque
anthony.ventresque@ucd.ie

CLI (Bash) for Big Data

School of Computer Science, UCD

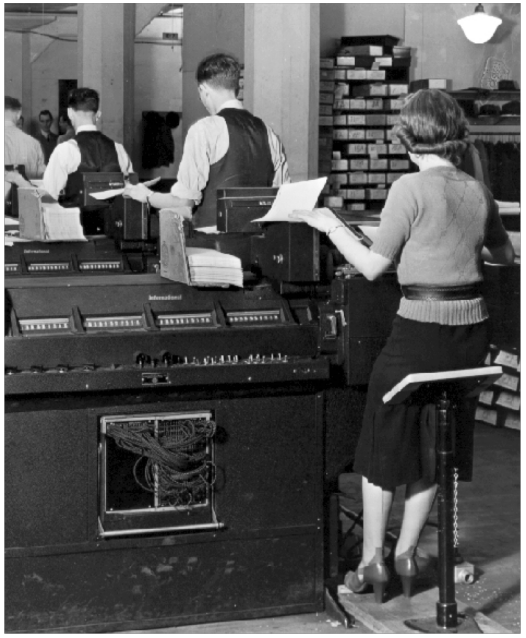
Scoil na Ríomheolaíochta, UCD

What Does an Operating System Do?

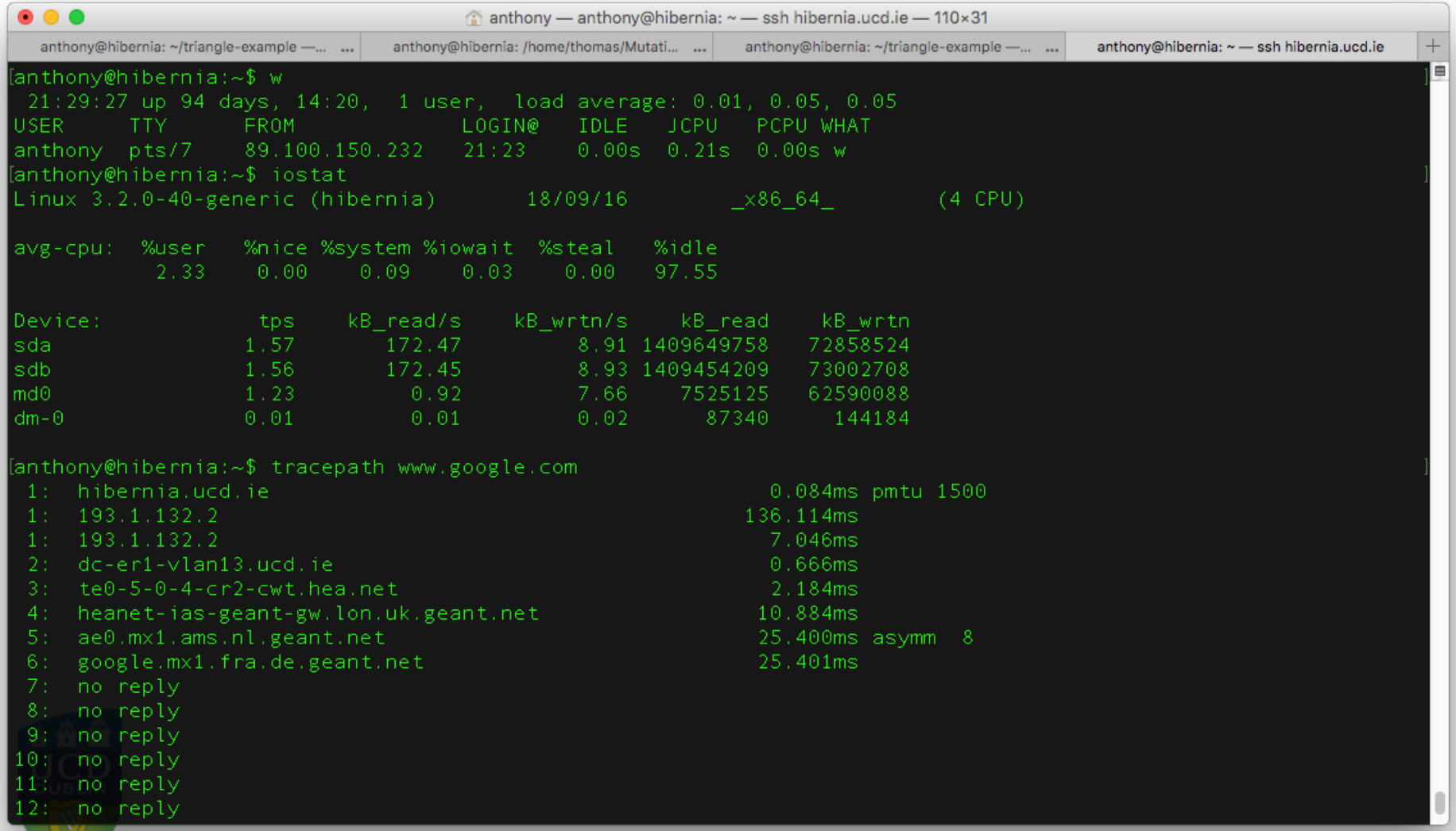
- An OS is a ***program*** that acts as an ***intermediary*** between a user of a computer and the computer hardware
- Goals: Execute user programs, make the computing system easy to use, utilise hardware efficiently
- OS is:
  - ***Resource allocator***: decides between conflicting requests for efficient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

User OS Interfaces

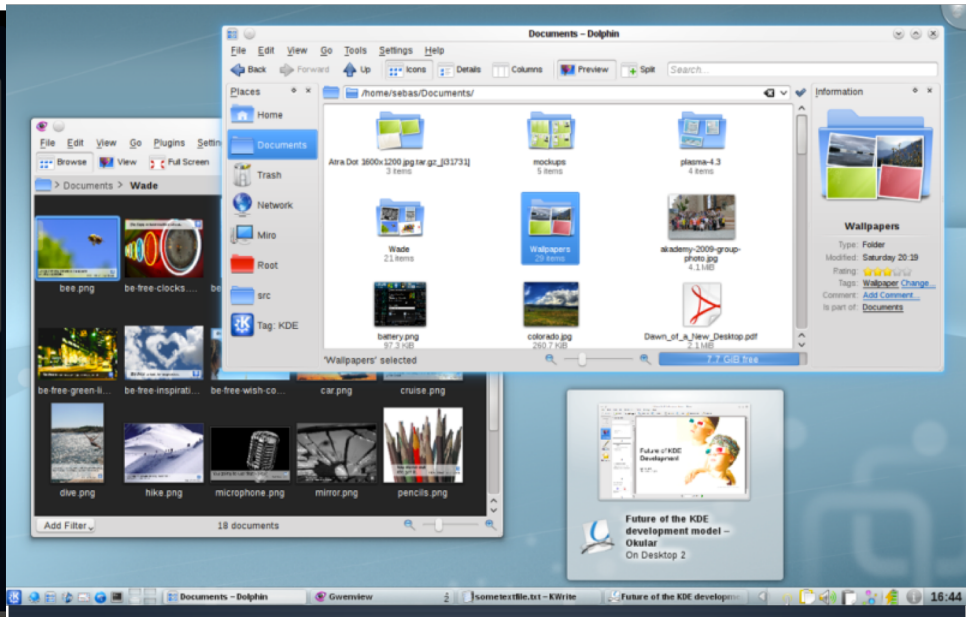
- Almost all operating systems have a user interface (UI).
  - ***Command-Line*** (CLI),
  - ***Graphics User Interface*** (GUI),
  - ***Batch*** (non-interactive, can be organised by job schedulers, workflow managers etc.)

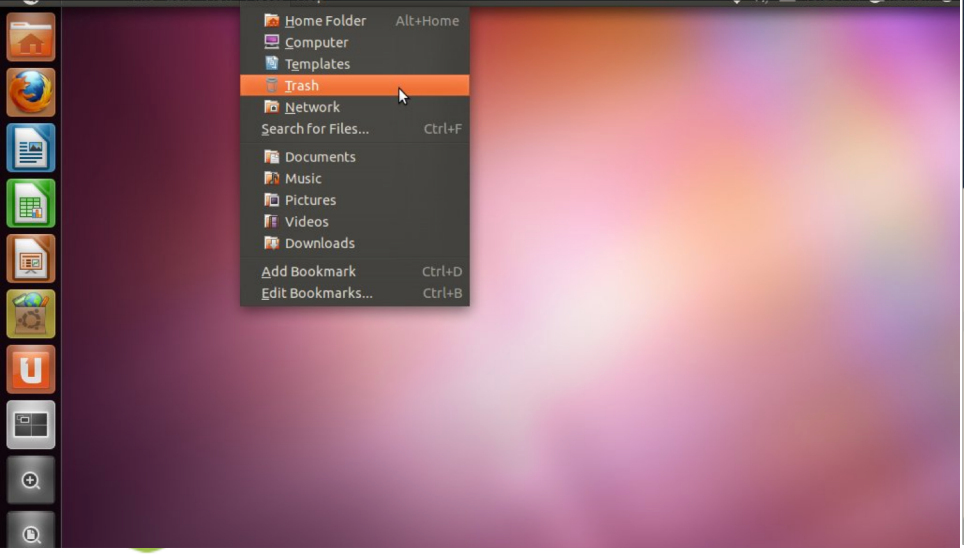


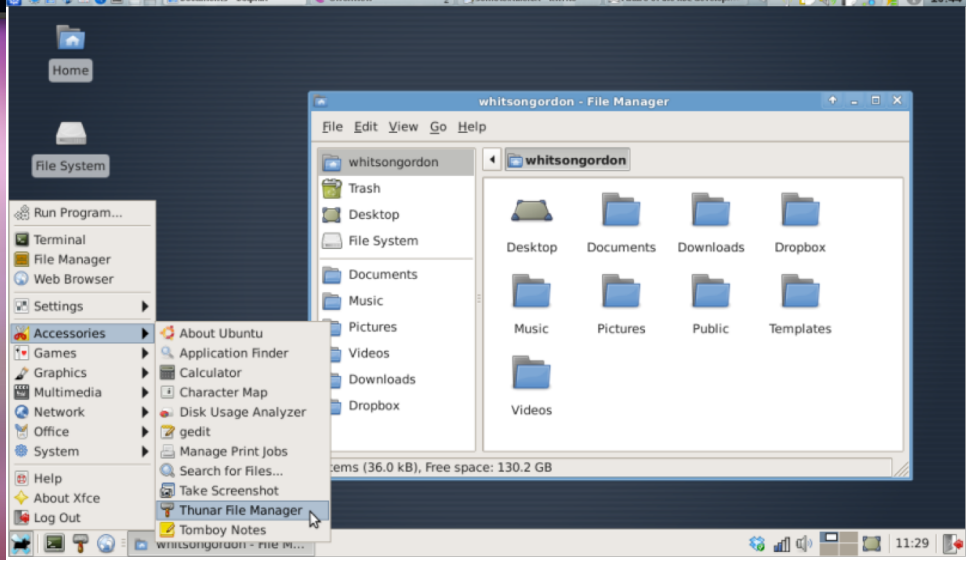
CLI: Bash - Bourne-again shell



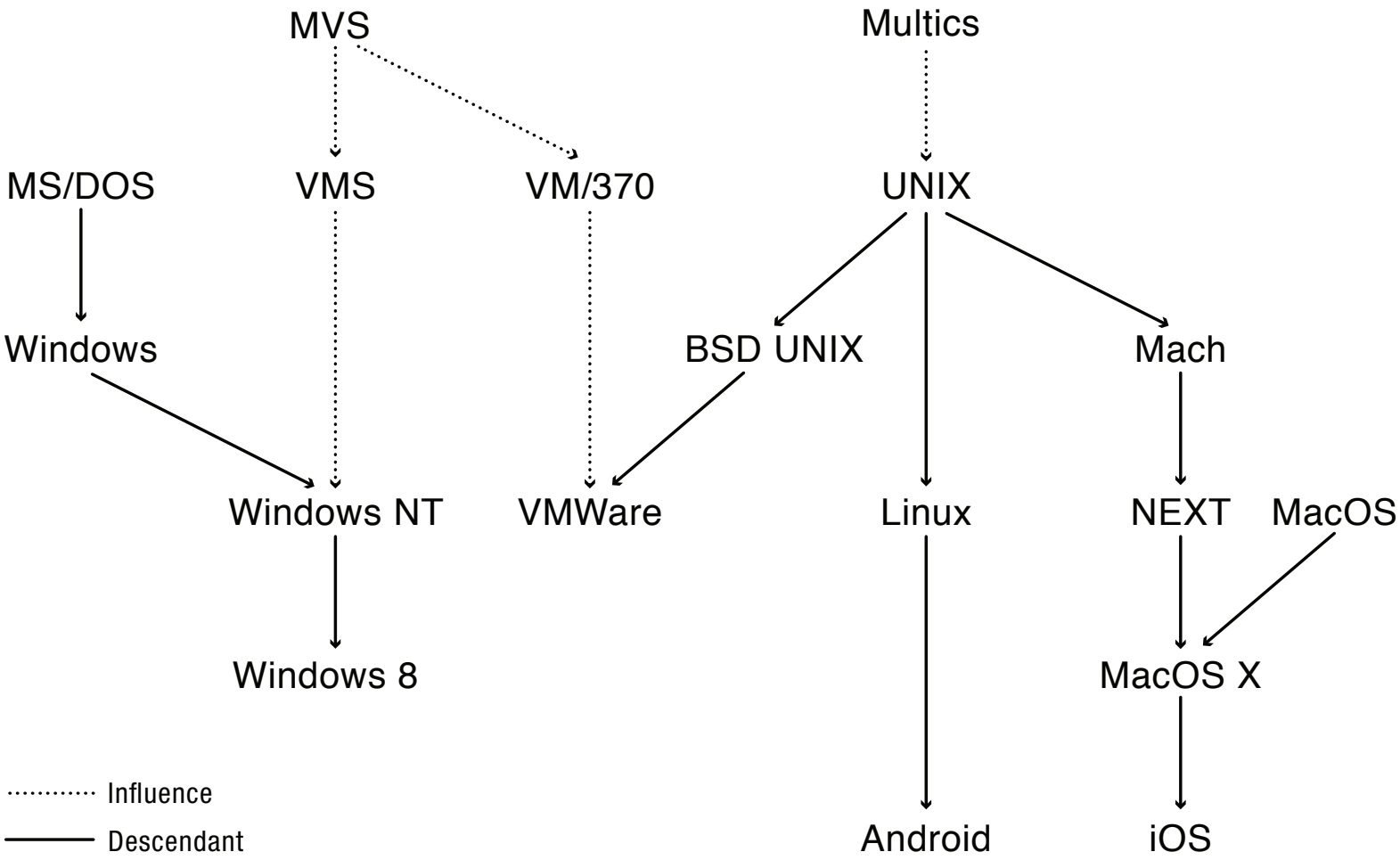
GUI: Gnome vs KDE vs Unity vs XFCE vs etc.



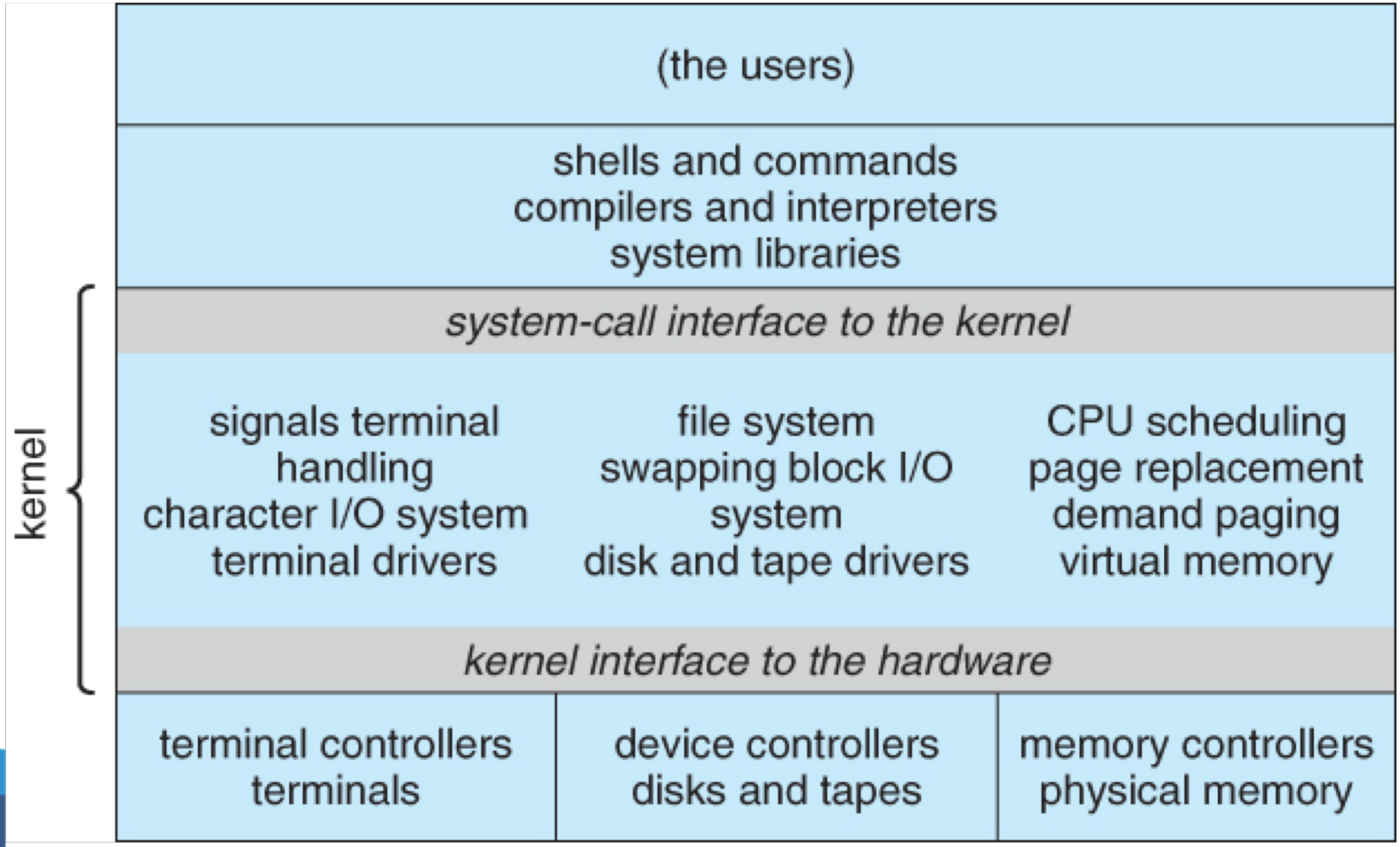




Desktop OS History



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

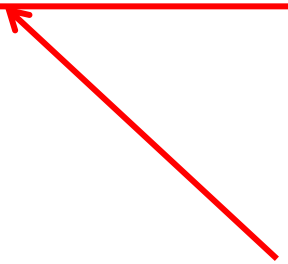
A Bash Text

Here we have 5 words

“In Bash this is a unique word”

Here, three, words

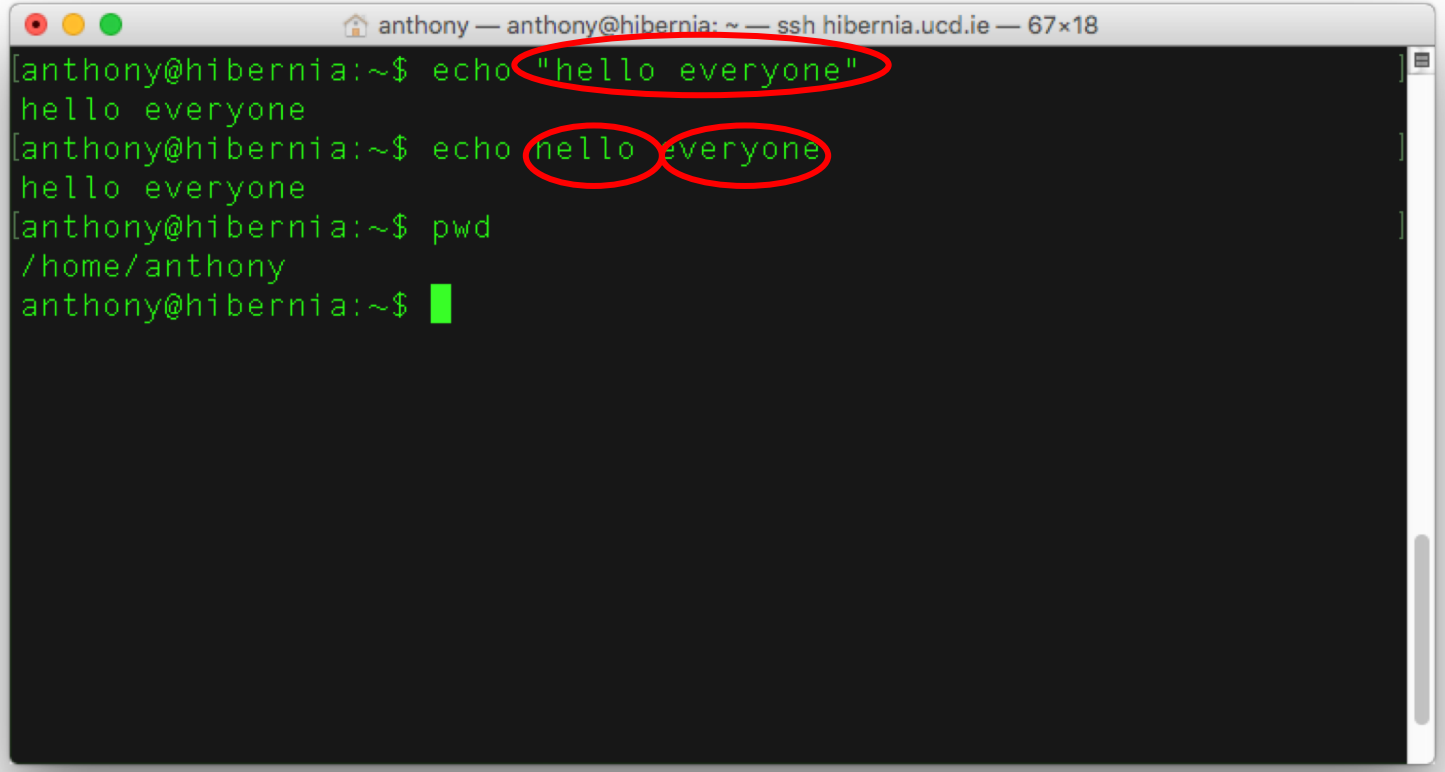
We|have;NINE&&words&here



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.

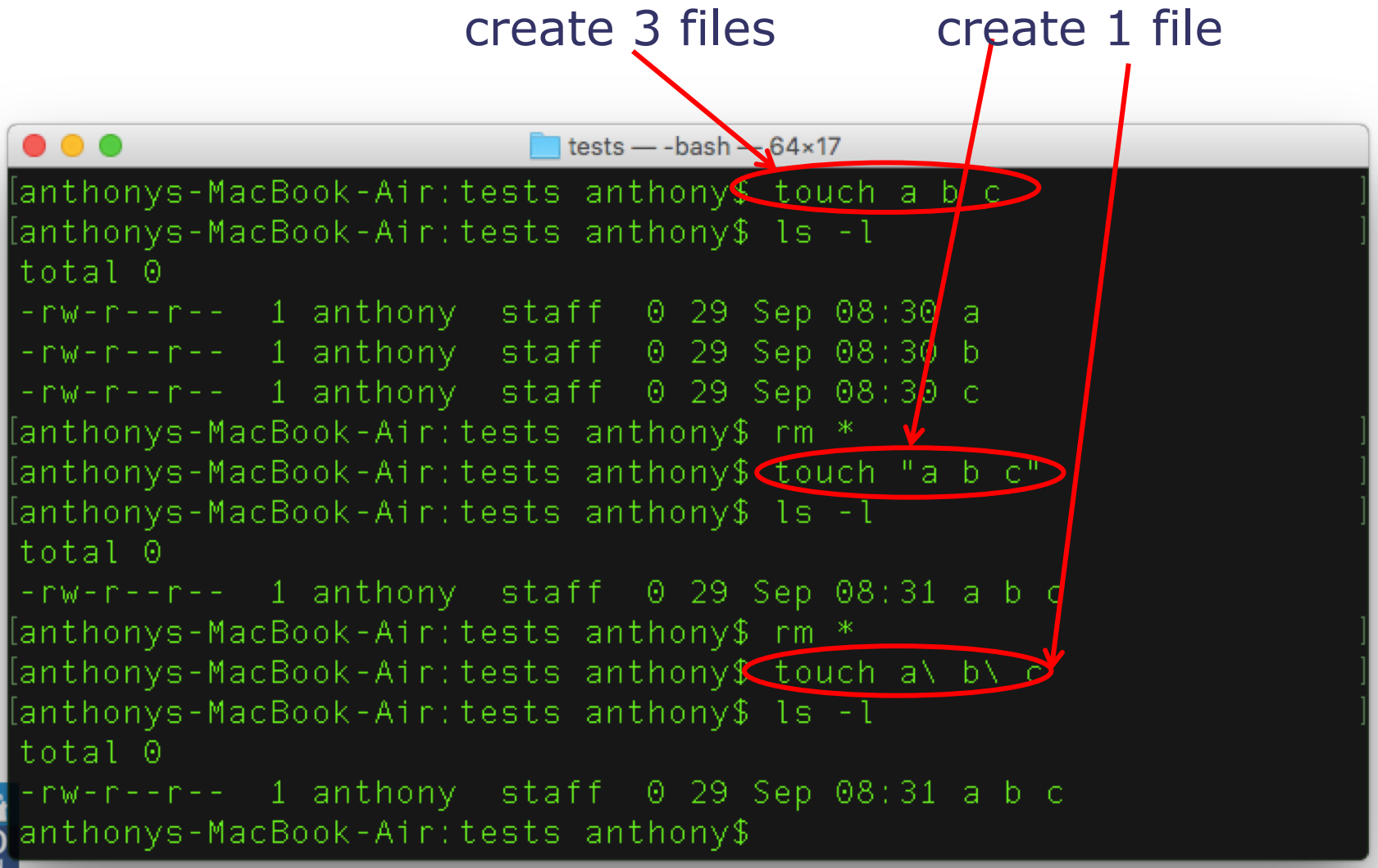


Special Characters

- 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

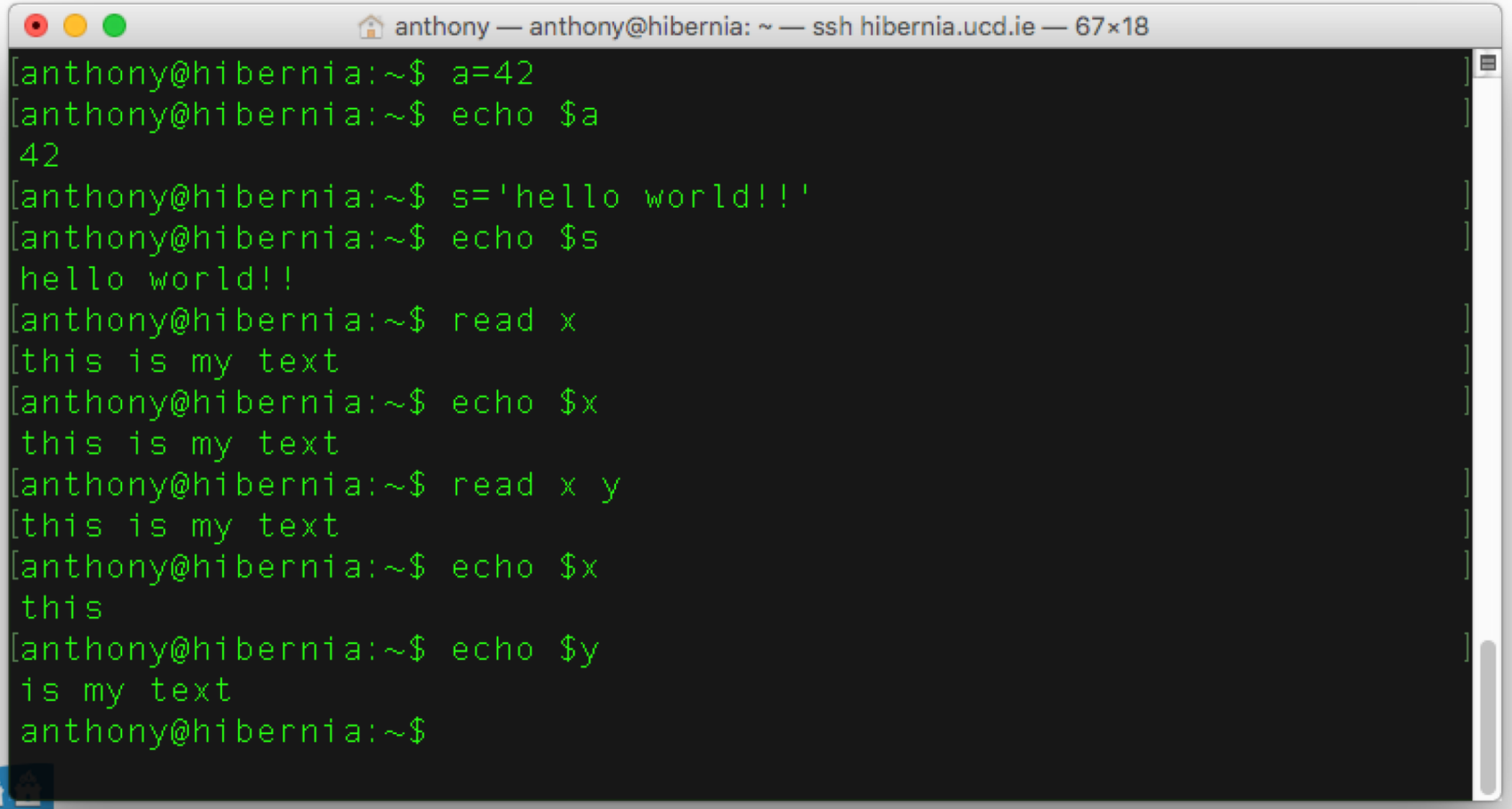
  – this gives the kernel the path to the command processor

```
#!/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



Bash = Imperative Programming

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

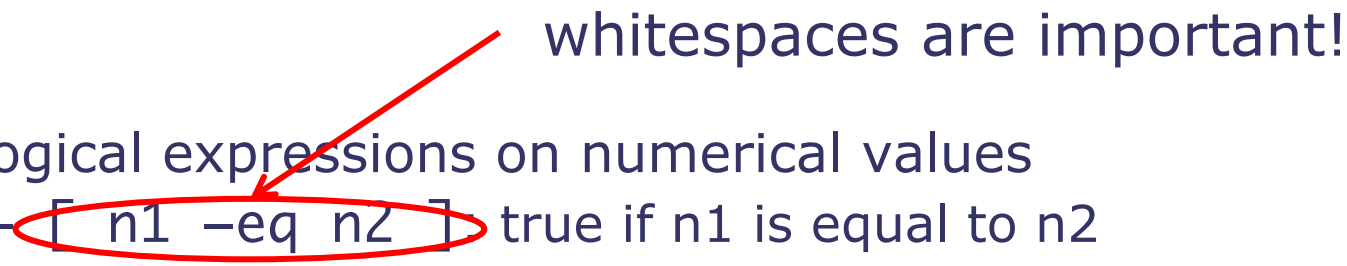
```
instruction1
instruction2
instruction3; instruction4
instruction5
```

Conditional Expression

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

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

Logical Expressions



- 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

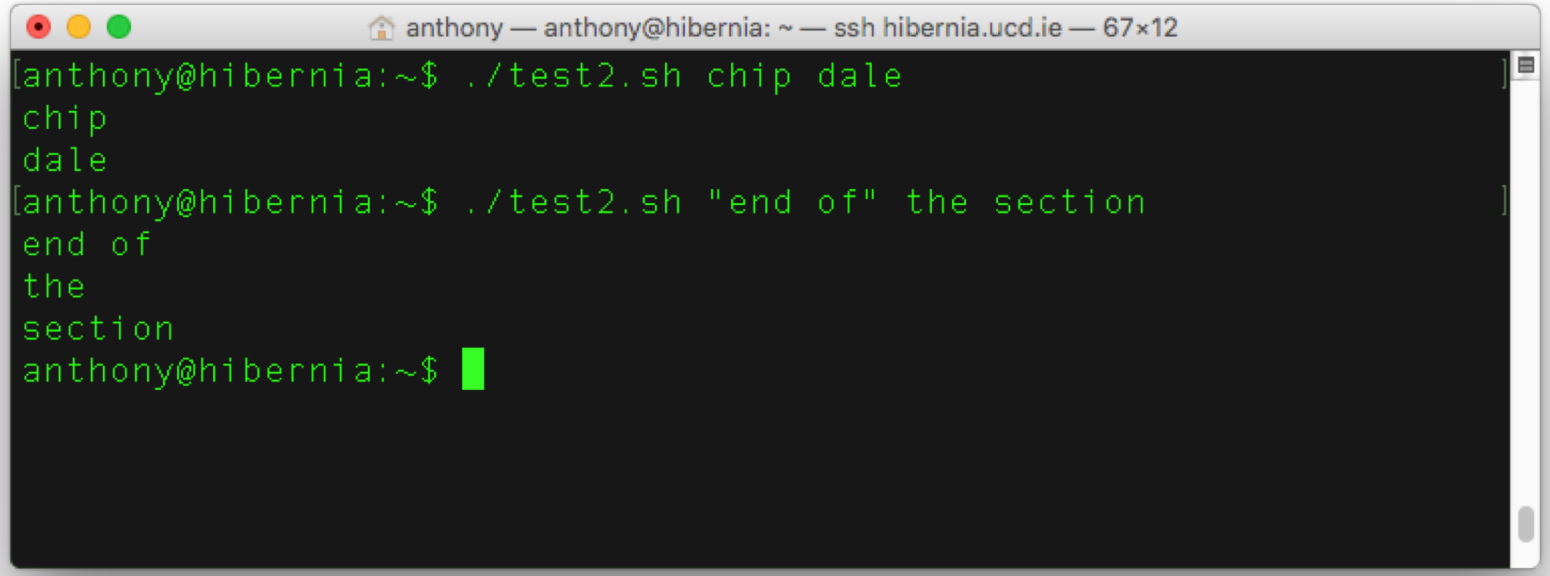
- ./my_script.sh arg1 arg2 arg3 …
- every word is stored in a variable

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

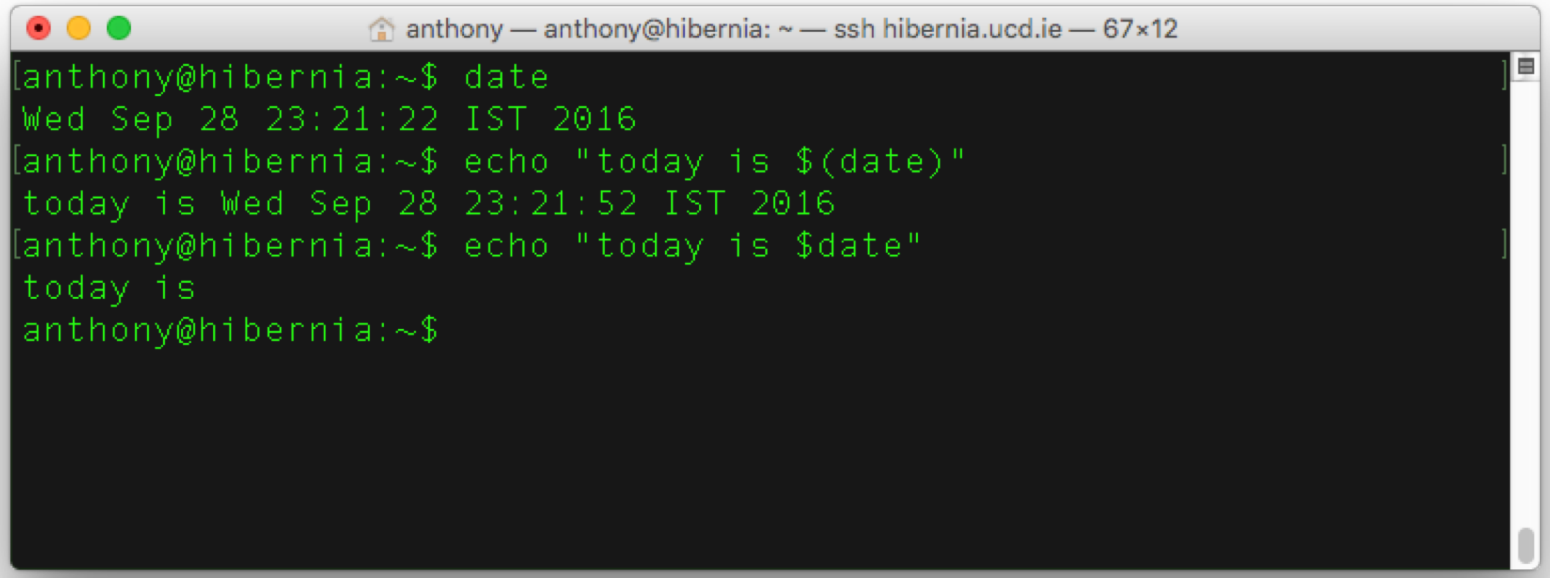
Example

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



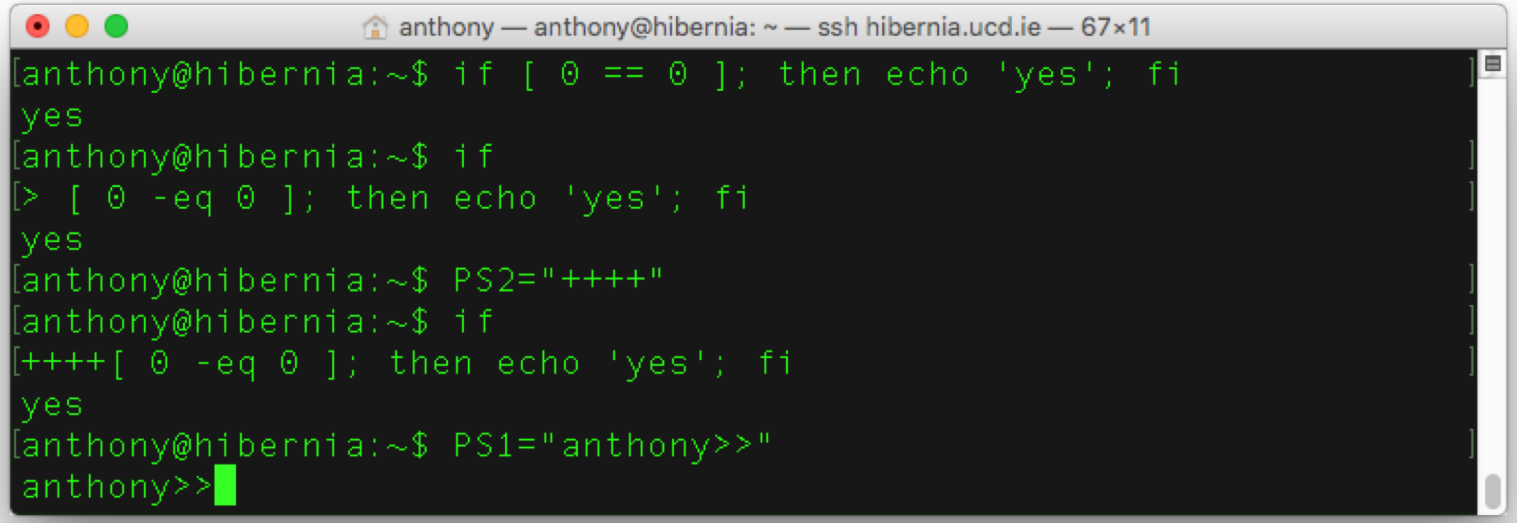
Nested Commands

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



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).

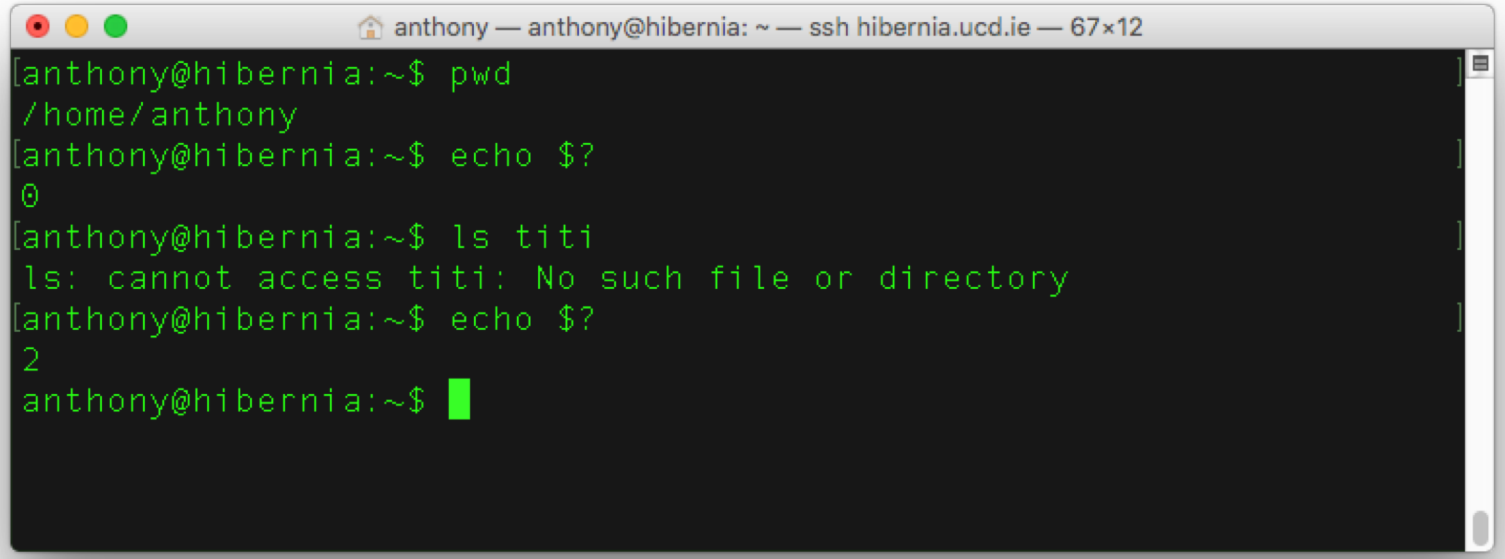


PATH

- 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…

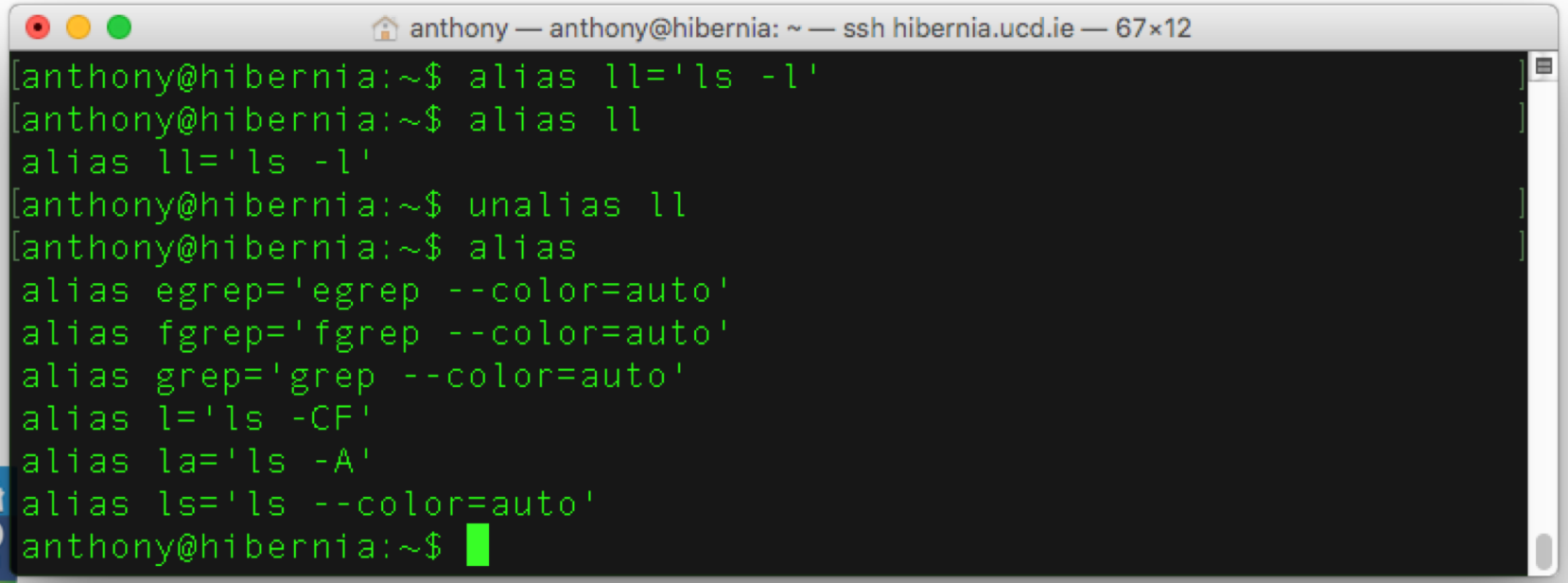
Exit Code

- 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 $?



Bash Alias

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



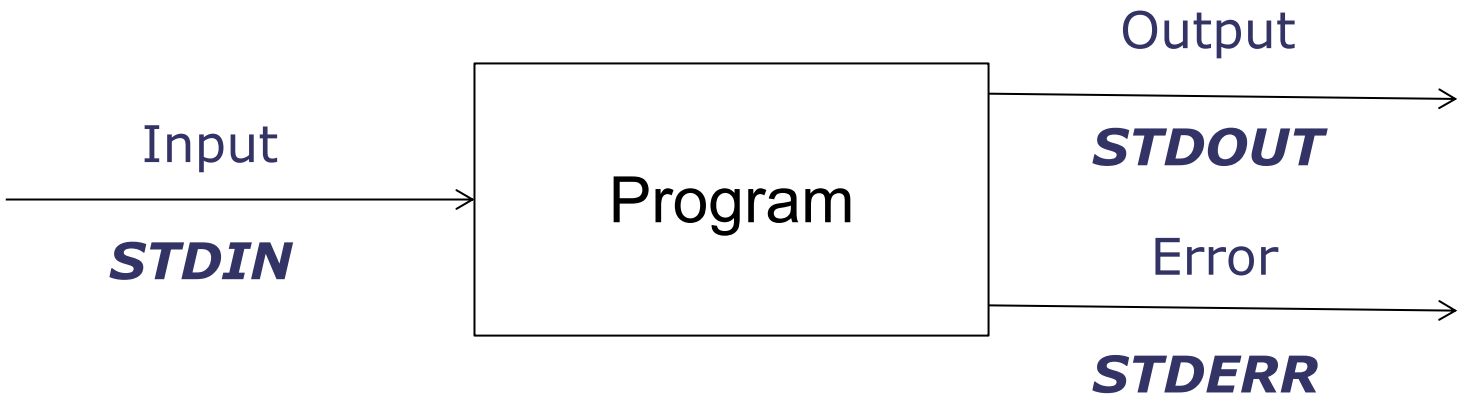
Configuration Files

- 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 seperate 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