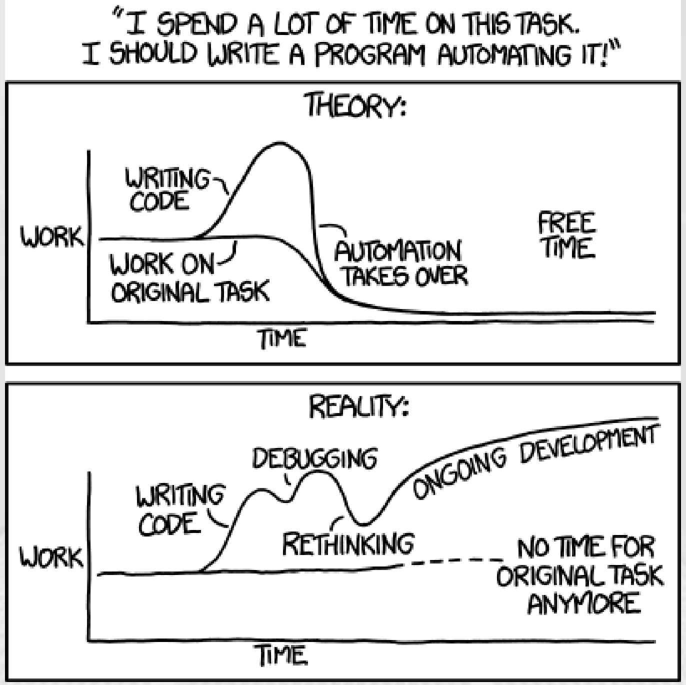
P2T 2025: Linux Lecture 4

Bash Scripting



https://xkcd.com/1319

Dr Gordon Stewart
Room 427, Kelvin Building
gordon.stewart@glasgow.ac.uk

Part 1:
Simple Scripts

What is a shell script?

- A **shell script** is a collection of commands to be run by a Linux shell's command-line interpreter
  - A shell script is just a simple program
  - Many of the concepts and structures that you will have encountered in C or Python also apply to shell scripts
- Shell scripts allow you to automate common or time-consuming tasks, for example:
  - Backing-up files from one location to another
  - Running raw data files through a series of processing steps to prepare them for later use
  - Checking log files and issuing alerts when certain errors are encountered
- Shell scripts are **interpreted**
  - They do not need to be compiled prior to being executed
  - A shell script can be executed directly, provided the execute permission is set:

```
chmod a+x myscript.sh
```

- This course focuses on Bash scripts, but other shells offer similar functionality

Other sources of information

- In P2T, we just cover the basics of shell scripting
  - The *Bash Scripting Guide* issued alongside labs 3 and 4 includes everything needed for P2T
- Some example scripts from this lecture are accessible from Brutha:
  - **`~/examples/p2t/linux/Lecture04`**
- The Linux Documentation Project has comprehensive guides if you want to find out more:
  - Bash Beginners' Guide: https://tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html
  - Advanced Bash Scripting: https://tldp.org/LDP/abs/html/index.html
- Sites such as stackoverflow.com can be helpful when trying to solve specific problems
  - In Linux, there is often more than one way to solve a problem, so solutions you find online may be quite different to those suggested in this course
- ***Please remember to cite your sources if using material from online guides***

Notes on the slides formatted like this contain information or explanations which may be helpful, but which are beyond the scope of P2T. These notes are not examinable!

A simple Bash script

~/examples/p2t/linux/Lecture04/script1.sh

```bash
#! /bin/bash

# This is a comment.  Anything after
# the '#' is ignored by Bash.

echo "The current directory is ${PWD}"
echo "It contains:" # This is also a comment
ls

# Commands can be combined on a single line
# by separating them with semicolons
whoami; pwd

# Sleep for a while
echo "Sleeping for 1 second..."
sleep 1
echo "Sleeping for 2 seconds..."
sleep 2
echo "I've woken up again"

# Exit with successful code
echo "Exiting"
exit 0
```

./script1.sh

```
[gordon@brutha0 scripts]$ ./script1.sh
The current directory is /home/gordon/scripts
It contains:
script1.sh
gordon
/home/gordon/scripts
Sleeping for 1 second...
Sleeping for 2 seconds...
I've woken up again
Exiting
[gordon@brutha0 scripts]$
```

The interpreter directive

- A Bash script is a text file containing a series of commands
- By convention, scripts have the **`.sh`** extension
- The first line beginning **`#!`** is the **interpreter directive**
  - Tells Linux which program to use to run the script
  - **`#!`** is actually a two-byte "magic number" which identifies the file as an executable script
  - Sometimes called the **shebang**, **sha-bang**, **hash-bang** or **hash-pling**…
  - Effectively this tells Linux to run:

    `/bin/bash script.sh`
- All Bash scripts should start with this line:

  `#! /bin/bash`

```
#! /bin/bash

# This is a comment.  Anything after
# the '#' is ignored by Bash.

echo "The current directory is ${PWD}"
echo "It contains:" # This is also a comment
ls

# Commands can be combined on a single line
# by separating them with semicolons
whoami; pwd

# Sleep for a while
echo "Sleeping for 1 second..."
sleep 1
echo "Sleeping for 2 seconds..."
sleep 2
echo "I've woken up again"

# Exit with successful code
echo "Exiting"
exit 0
```

Comments

- As in C, it may be useful to include comments in your code to help a reader understand it
  - Emphasis not on *what* the code does, but *why*
- A comment in Bash begins with the hash (**#**) character
  - Comments can begin at the start of a line, or in the middle
  - Everything after the **#** is ignored
- Comments cannot come before the interpreter directive
- ***Please remember to cite your sources if using material from elsewhere***

```
#! /bin/bash

# This is a comment.  Anything after
# the '#' is ignored by Bash.

echo "The current directory is ${PWD}"
echo "It contains:" # This is also a comment
ls

# Commands can be combined on a single line
# by separating them with semicolons
whoami; pwd

# Sleep for a while
echo "Sleeping for 1 second..."
sleep 1
echo "Sleeping for 2 seconds..."
sleep 2
echo "I've woken up again"

# Exit with successful code
echo "Exiting"
exit 0
```

Exit codes

- In Linux, programs and scripts can return a numerical value when they finish which lets them report success or failure
  - In C, **`return 0`** at the end of the **`main`** function
- **`0`** is used to indicate success, and any non-zero value indicates a lack of success (e.g. a problem or error)
  - Different return codes can be used to distinguish between different outcomes
  - Usually documented in a command's manual pages
  - In Linux, the acceptable range is **`0`** to **`255`**
- In Bash, we can **`exit`** with a return code using the exit command:

```
exit 123
```

- We can check this value using the special variable **`$?`**

```
#! /bin/bash

# This is a comment.  Anything after
# the '#' is ignored by Bash.

echo "The current directory is ${PWD}"
echo "It contains:" # This is also a comment
ls

# Commands can be combined on a single line
# by separating them with semicolons
whoami; pwd

# Sleep for a while
echo "Sleeping for 1 second..."
sleep 1
echo "Sleeping for 2 seconds..."
sleep 2
echo "I've woken up again"

# Exit with successful code
echo "Exiting"
exit 0
```

Running a script

- Several common ways to run a script:

  `source script.sh`

  `chmod a+x script.sh` then **`./script.sh`**

- **`source`** runs the commands one by one in the current shell
  - The script does not need to be executable
  - This can be used to modify the running environment
  - This can be dangerous as variables in your current session may be overwritten, and an **`exit`** command will end your current session
- Making the script executable with **`chmod`** and running it as **`./script.sh`** forks a new process

```bash
#! /bin/bash

# This is a comment.  Anything after
# the '#' is ignored by Bash.

echo "The current directory is ${PWD}"
echo "It contains:" # This is also a comment
ls

# Commands can be combined on a single line
# by separating them with semicolons
whoami; pwd

# Sleep for a while
echo "Sleeping for 1 second..."
sleep 1
echo "Sleeping for 2 seconds..."
sleep 2
echo "I've woken up again"

# Exit with successful code
echo "Exiting"
exit 0
```

You may also see **`.`** used as an alternative to **`source`** online, but we will not use this in P2T

A simple Bash script: what does it do?

- **`~/examples/p2t/linux/Lecture04/script1.sh`**
- **`echo`** is used to print out text and the value of variables
- Invokes **`ls`** to print out a directory listing
- The commands **`whoami`** and **`pwd`** are given on a single line, separated by a semicolon (**`;`**)
- Uses the **`sleep`** command to pause for a number of seconds
- Uses the **exit** command to return the value 0
  - **`exit`** terminates the script and returns a numerical value to the calling process
  - **`0`** is used to indicate success, and any non-zero value is used to indicate an error of some sort

```
#! /bin/bash

# This is a comment.  Anything after
# the '#' is ignored by Bash.

echo "The current directory is ${PWD}"
echo "It contains:" # This is also a comment
ls

# Commands can be combined on a single line
# by separating them with semicolons
whoami; pwd

# Sleep for a while
echo "Sleeping for 1 second..."
sleep 1
echo "Sleeping for 2 seconds..."
sleep 2
echo "I've woken up again"

# Exit with successful code
echo "Exiting"
exit 0
```

Case study: data logger

- Imagine a **`datalogger`** program exists which reads values from a number of sensors and stores these readings in a log file
  - Occasionally the sensors go offline, in which case it returns empty readings
- We want to run this program periodically in order to record the sensor readings while an experiment is taking place
- After running the program, we need to tidy up the results files
- To simplify things, we're going to automate this activity by writing a script which:
  - Lets you specify which datalogger you want to use and checks it is executable
  - Runs the datalogger a given number of times at an interval of a couple of seconds
  - Removes any log files produced while the sensors were offline
  - Prints out the readings

```
[gordon@brutha2 l4-bash]$ ./datalogger
[gordon@brutha2 l4-bash]$ cat 20230131-121424.dat
TIMESTAMP                  TEMP   HUMID  STATUS
2023-01-31T12:14:19.989931 24.077 48.456 Online
[gordon@brutha2 l4-bash]$ ./datalogger
[gordon@brutha2 l4-bash]$ cat 20230131-121424.dat
TIMESTAMP                  TEMP   HUMID  STATUS
2023-01-31T12:14:24.070543 ------ ------ Offline
```

Part 2: Variables

Variables

- You can set a variable like this:

  `variable=value`

- Variable names are case-sensitive
- You can store the output of a command in a variable using **command substitution**:

  `variable=$(command)`

> An older syntax enclosing the command in backticks (`) also exists: **`` variable=`command` ``**

- Bash has two types of string: single-quoted (') and double-quoted (")
  - In double-quoted strings, variable names are substituted with their value
  - In single-quoted strings, variable substitution does not take place
  - ***This is different to C!***

~/examples/p2t/linux/Lecture04/variables.sh

```
#! /bin/bash

# Script demonstrating various uses
# of variables

foo=12
bar=3
echo "foo = ${foo}, bar = ${bar}"

# Command substitution
my_name=$(whoami)
echo ${my_name}

# Strings
echo "This is a double-quoted string which refers to ${PWD}"
echo 'This is a single-quoted string which refers to ${PWD}'

exit 0
```

Special variables

- Bash has some special variables that provide useful information
- Access command-line arguments passed to the script:
  - **$0** is the name of the script (similar to **argv[0]** in C)
  - **$1**, **$2**... are the first, second, and subsequent arguments (similar to **argv[1]** etc. in C)
  - **$#** is the number of arguments (excluding **$0**)
  - **$@** contains all arguments, and can be useful in a **for** loop (excluding **$0**)
- Get information about a script or command:
  - **$$** contains the process ID (PID) of the script
  - **$?** contains the exit code of the last command, which can be used to check whether it was successful

Variable	Description
$0	The name of the script
$#	The number of command-line arguments passed to the script
$1, **$2**, etc.	The first (**$1**), second (**$2**) and subsequent arguments passed to the script
$@	All the arguments passed to the script
$$	The process ID (PID) of the script
$?	The exit code of the last command

Example: special variables

- `~/examples/p2t/linux/Lecture04/specvar.sh`
- This example:
  - Prints the name and PID of the script
  - Prints the number of arguments, and the first and second argument
  - Loops over all arguments, and prints these out one at a time
  - Runs a series of different commands to demonstrate different return codes – check manual pages or program documentation to find out what these mean
- `/dev/null` is a special file which can be used in redirection to dispose of a stream (i.e. it throws away the output)

```bash
#! /bin/bash

# Examples demonstrating special Bash variables

# Name and PID
echo "This is ${0} with PID $$"

# Arguments
echo "Number of arguments: $#"
echo "- Argument 1: ${1}"
echo "- Argument 2: ${2}"

# Loop over arguments
for arg in $@; do
    echo $arg
done

# Testing return codes
echo "Return codes:"
ls > /dev/null
echo "ls: $?"
clang > /dev/null 2>&1
echo "clang: $?"
clang broken.c > /dev/null 2>&1
echo "clang broken.c: $?"

exit 0
```

Part 3: Conditional Statements and Tests

Conditional statement: `if`

- The conditional **`if`** statement in Bash is similar to that in C or Python
  - Allows a particular code branch to be executed if certain conditions are met
  - For example, **if** a command returns an error code **then** print a warning and exit, **else** continue to run as normal
- The general form of the **`if`** statement is shown to the right
  - You can have zero or more **`elif`** branches to test additional conditions
  - You can have zero or one **`else`** branch to perform some default action
- As with commands, the **`then`** keyword can appear on the same line as the **`if`** or **`elif`** statement if separated by a semicolon:

```
if condition; then
```

```
if condition
then
    do something
elif condition
then
    do something different
else
    do something else
fi
```

Conditional tests

- A conditional test looks like this:

```
[[ expression ]]
```

- The spaces between the square brackets and the expression are important!
- You can combine tests using **&&** for "and" and **||** for "or", for example:

```
[[ -e foo.c && -w foo.c ]]
```

Older syntaxes using **[ ]** or **test** also exist – see note in lab 3

String comparison	Result
`string1 == string2`	True if the strings are equal
`string1 != string2`	True if the strings are not equal
`-n string1`	True if the length of the string is not zero
`-z string1`	True if the length of the string is zero

Arithmetic tests	Result
`A -eq B`	True if the expressions are equal
`A -ne B`	True if the expressions are not equal
`-gt`, `-ge`, `-lt`, `-le`	Used as above: true if the first expression is greater than (`-gt`), greater than or equal to (`-ge`), less than (`-lt`), or less than or equal to (`-le`), the second
`!A`	Inverts the expression: true if the expression is false, or vice-versa

File tests	Result
`-d FILENAME`	True if the file is a directory
`-e FILENAME`	True if the file exists
`-f FILENAME`	True if the file is a regular file
`-r FILENAME`	True if the file is readable
`-s FILENAME`	True if the file is not empty
`-w FILENAME`	True if the file is writable
`-x FILENAME`	True if the file is executable

Example: conditionals

- **`~/examples/p2t/linux/Lecture04/conditional.sh`**
- The script begins with the mandatory interpreter directive, followed by a header comment which describes its purpose
- Command substitution is used to extract the username of the user running the script
- A string comparison test is performed to check if the script is running as root, and it exits with an error if this is the case
- A conditional statement is used to check if one of various directories exists, and an error is displayed if none of these exist
- The contents of this directory is listed
- The script exits with a return code of **`0`** indicating success

```
#! /bin/bash

# Script demonstrating conditional statements.
#
# Returns:
#   0  Success
#   1  Run as root
#   2  No output directory

# Check which user is running the script
script_user=$(whoami)

# Don't run this as root!
if [[ ${script_user} == "root" ]]; then
    echo "ERROR: Do no run this script as root"
    exit 1
fi

# Decide which directory to use
if [[ -d output ]]; then
    target="output"
elif [[ -d results ]]; then
    target="results"
else
    echo "ERROR: Target directory does not exist"
    exit 2
fi

# List directory contents
echo "Directory ${target} contains:"
ls -l $target

# Exit with successful status code (0)
exit 0
```

Part 4: Ranges and Lists (and Arrays)

Ranges

- We often want to loop over ranges of values in scripts
- A numerical range can be written as **`{first..last}`**:

  `{1..3}` → 1 2 3

  `{03..01}` → 03 02 01

- An increment can also be provided:

  `{1..9..2}` → 1 3 5 7 9

- A range can use characters:

  `{A..C}` → A B C

- We cannot include variables when using this notation
- The **`seq`** command can be used to generate numerical ranges, and can be combined with variables:

```
end=3
seq 1 ${end}
```

```
[gordon@brutha0 ~]$ echo {1..10}
1 2 3 4 5 6 7 8 9 10
[gordon@brutha0 ~]$ echo {10..01}
10 09 08 07 06 05 04 03 02 01
[gordon@brutha0 ~]$ echo file{001..003}.dat
file001.dat file002.dat file003.dat
[gordon@brutha0 ~]$ echo {A..G}
A B C D E F G
[gordon@brutha0 ~]$ end=6
[gordon@brutha0 ~]$ echo {3..${end}}
{3..6}
[gordon@brutha0 ~]$ seq 3 ${end}
3
4
5
6
[gordon@brutha0 ~]$ seq -s ' ' 3 ${end}
3 4 5 6
```

Lists

- Bash treats a group of space-separated strings as a list
- You can also specify a list using braces, similar to a range:

  `{a,b,c,d}`

  - Be careful not to include spaces within the braces!
- Bash treats a single- or double-quoted string as a list if it contain spaces:

  `list1='a b c d'`

  `list2="a b c d"`

- This can be useful when processing the output of file system commands such as **`ls`**:

  `text_files=$(ls *.txt)`

```
[gordon@brutha0 scripts]$ mkdir example
[gordon@brutha0 scripts]$ touch example/data{A,B,C}.txt
[gordon@brutha0 scripts]$ ls example/
dataA.txt  dataB.txt  dataC.txt
[gordon@brutha0 scripts]$ list1="A B C D"
[gordon@brutha0 scripts]$ echo ${list1}
A B C D
[gordon@brutha0 scripts]$ list2='a b c d'
[gordon@brutha0 scripts]$ echo ${list2}
a b c d
```

Arrays

- Newer versions of Bash support arrays which can be accessed via indices
  - "Newer" is relative: Bash 2.0 introduced arrays in 1996!
- Create an array by enclosing space-separated values in brackets:

  `my_array=(a b c d)`
- Access an array using a zero-based index as in C or Python:

  `echo ${my_array[1]}`
- You can access all elements by using **@** as the index:

  `echo ${my_array[@]}`
- Array indices do not need to be consecutive
- Even newer (post 2009!) versions of Bash support associative arrays, similar to dictionaries in Python

```
[gordon@brutha0 ~]$ my_array=(a b c d)
[gordon@brutha0 ~]$ echo ${my_array[0]}
a
[gordon@brutha0 ~]$ echo ${my_array[1]}
b
[gordon@brutha0 ~]$ echo ${my_array[2]}
c
[gordon@brutha0 ~]$ echo ${my_array[3]}
d
[gordon@brutha0 ~]$ echo ${my_array[4]}

[gordon@brutha0 ~]$ my_array[5]=f
[gordon@brutha0 ~]$ echo ${my_array[4]}

[gordon@brutha0 ~]$ echo ${my_array[5]}
f
[gordon@brutha0 ~]$ echo ${my_array[6]}
```

Part 5:
for Loops

Loops: for

- The **for** loop is used when the number of items you wish to iterate over is known in advance
- Items can be values from a range, list or array, the results of running a command, or files from a glob:

```
x in {3..9}
file in *.pdf
end_text in $(tail filename.txt)
```

A C-style for loop also exists, but is rarely used:

```
for (( x = 0; x < 5; x++ ))
```

```
for expression
do
    do something
done
```

```
#! /bin/bash

# Examples of for loops

# Loop over a range
for number in {1..3}; do
    echo ${number}
done

# Loop over files matching a pattern
for text_file in docs/*.txt; do
    wc -l ${text_file}
done

# Loop over the output of a command
# (Here, the three most recently-modified files
#  in my home directory)
for recent in $(ls -t ~/ | head -n 3); do
    file ~/${recent}
done

exit 0
```

Example: `for`

- **`~/examples/p2t/linux/Lecture04/for.sh`**
- The first example loops over values in a range
  - This is the best way to replicate the C-style **`for`** loop
  - Any range is acceptable: it does not have to be numerical
- The second example loops over all files with names matching a particular pattern
  - We could loop over all files with a simpler glob:

    `for file in *; do`
- The third example loops over the output of another command, line by line
  - We can then run other commands on this output

```
for expression
do
    do something
done
```

```
#! /bin/bash

# Examples of for loops

# Loop over a range
for number in {1..3}; do
    echo ${number}
done

# Loop over files matching a pattern
for text_file in docs/*.txt; do
    wc -l ${text_file}
done

# Loop over the output of a command
# (Here, the three most recently-modified files
#  in my home directory)
for recent in $(ls -t ~/ | head -n 3); do
    file ~/${recent}
done

exit 0
```

Summary

- A **shell script** is a collection of commands to be run in a shell
- Shell scripts allow you to automate common or time-consuming tasks
- Shell scripts are **interpreted** and do not need to be compiled before execution
- Shell scripts begin with an **interpreter directive** which tells Linux which command interpreter to use
  - All Bash scripts begin **`#! /bin/bash`**
- Comments are introduced using the **`#`** character
- Bash has two types of string:
  - Variable substitution does not take place in single-quoted strings
  - Variable substitution does take place in double-quoted strings
- Special variables: **`$0`**, **`$1`**, **`$2`**, **`$@`**, **`$#`**, **`$$`**, **`$?`**
- Bash has conditional statements (**`if`**) for branching and loops (**`for`**) for repetition
  - Various tests can be used with these constructs
- Values can be collected together and iterated over: ranges and lists

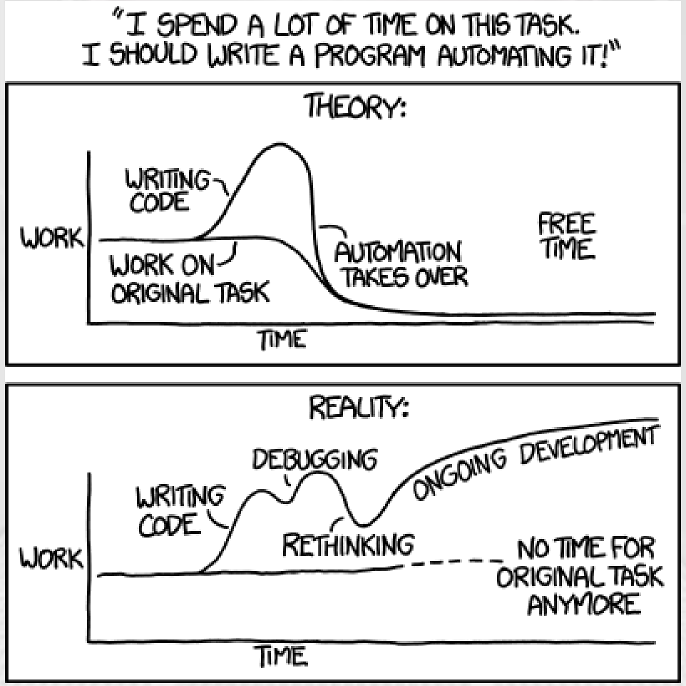
Case study: final script

- Our completed data logger script might look something like this…
  - …or it might not, depending on what I do in the lecture!
- Requirements
  - Lets you specify which datalogger you want to use and checks it is executable
  - Runs the datalogger a given number of times at an interval of a couple of seconds
  - Removes any log files produced while the sensors were offline
  - Prints out the readings

```
#! /bin/bash

# Script to process output from data logger.
# P2T Linux Lecture 4
# Gordon Stewart (30 January 2023)

# Get datalogger application
target=$1

# Check we have enough arguments
if [[ $# != 1 ]]; then
    echo "Usage: ./example.sh APPLICATION"
    exit 1
# Check argument is executable
elif [[ ! -x ${target} ]]; then
    echo "ERROR: Not executable: ${target}" >&2
    exit 2
fi

# Run datalogger
for runcount in {1..10}; do
    echo "Running datalogger: ${runcount}"
    ${target}
    sleep 2
done

# Check data files
for filename in *.dat; do
    # Remove offline readings / print data
    grep 'Offline' ${filename} > /dev/null
    if [[ $? == 0 ]]; then
        echo "Removing offline data file: ${filename}"
        rm -f ${filename}
    else
        echo "DATA: $(tail -n 1 ${filename})"
    fi
done

# Success!
exit 0
```

P2T 2025: Linux Lecture 4

Bash Scripting



https://xkcd.com/1319

Dr Gordon Stewart
Room 427, Kelvin Building
gordon.stewart@glasgow.ac.uk