CT213

Bash Scripts

Takfarinas Saber
takfarinas.saber@universityofgalway.ie

Structure of a Bash Script

- First line always contains the following:

  #!/bin/bash

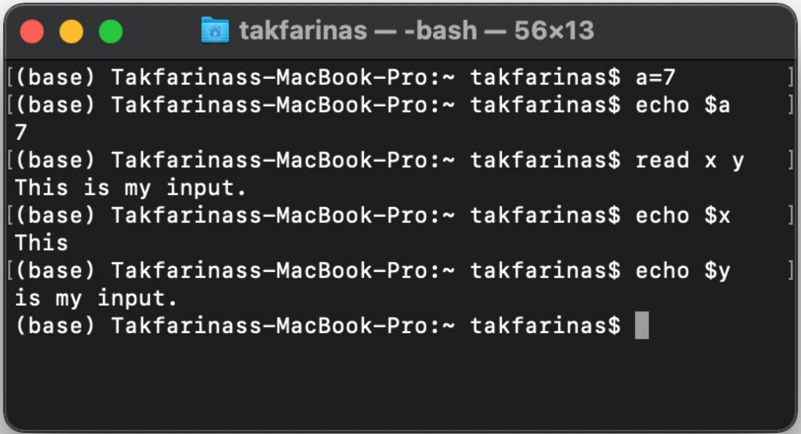
– this gives the kernel the path to the command processor

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

Bash Variables

- Assign a variable using "=": `my_var = value`
  - e.g., `a=7`
- $ is used to read a variable: `$my_var`
  - e.g, `echo "a"`
- interactive read: `read var1 var2 ...`

  - reads some input given by the user (until new line character)
  - first word in var1, etc.
  - e.g., read x y



```
[(base) Takfarinass-MacBook-Pro:~ takfarinas$ a=7                  ]
[(base) Takfarinass-MacBook-Pro:~ takfarinas$ echo $a              ]
7
[(base) Takfarinass-MacBook-Pro:~ takfarinas$ read x y             ]
This is my input.
[(base) Takfarinass-MacBook-Pro:~ takfarinas$ echo $x              ]
This
[(base) Takfarinass-MacBook-Pro:~ takfarinas$ echo $y              ]
is my input.
(base) Takfarinass-MacBook-Pro:~ takfarinas$ 
```

Bash is an Imperative Programming

Sequence of instructions

- on different lines
- or separated by “;”

```
Instruction1
Instruction2
instruction3; instruction4
instruction5
```

Conditional Expressions

```
if cond1; then
    instructions
elif cond2; then
    instructions
else instructions
fi
```

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

Logical Expressions

Spaces 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
- `[ n1 –lt n2 ]`: true if n1 n2 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

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
```

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

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

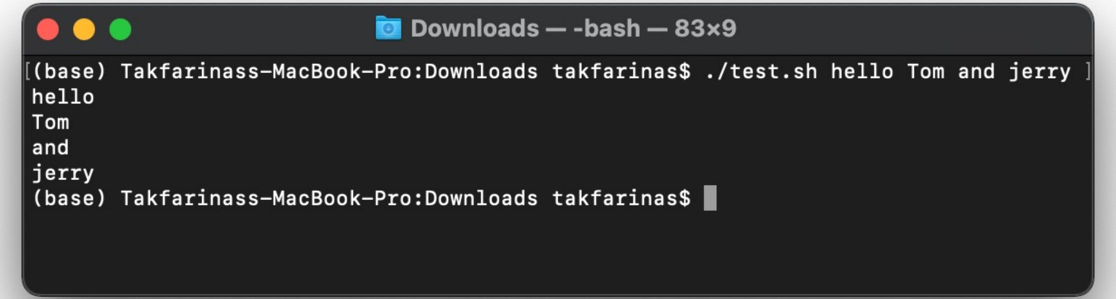
my_script.sh	arg1	arg2	arg3	arg4	...
"$0"	”$1"	”$2”	”$3”	”$4"	…

- "$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
```



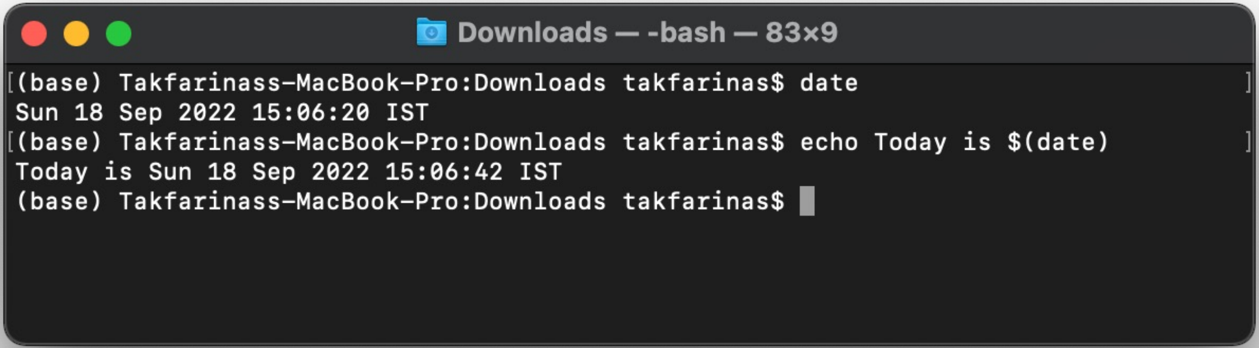
Downloads — -bash — 83×9

```
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ ./test.sh hello Tom and jerry
hello
Tom
and
jerry
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ 
```

It will print all the parameters (including the name of the script).

Nested Commands

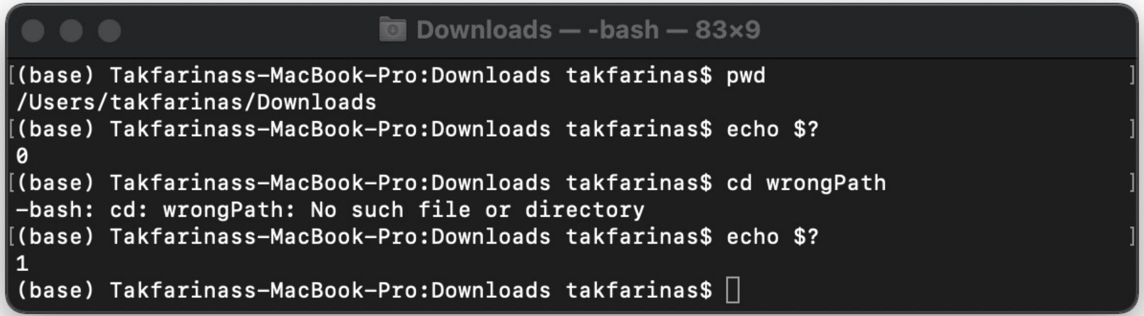
- To gets the (text) output of a command cmd, use $(cmd)
  – different from $cmd (gives access to variable cmd not command)



```
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ date
Sun 18 Sep 2022 15:06:20 IST
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ echo Today is $(date)
Today is Sun 18 Sep 2022 15:06:42 IST
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ 
```

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 $?



```
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ pwd
/Users/takfarinas/Downloads
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ echo $?
0
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ cd wrongPath
-bash: cd: wrongPath: No such file or directory
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ echo $?
1
(base) Takfarinass-MacBook-Pro:Downloads takfarinas$ 
```

Arithmetic Operations

In bash, the command that you should use to play with arithmetic expressions is **expr**

```
$> expr 1 + 1
2
$> myvar=$(expr 1 + 1)
$> echo $myvar
2
$> expr $myvar + 1
3
$> expr $myvar / 3
1
$> expr $myvar \* 3
9
Etc.
```

CT213
Computing Systems & Organisation

University of Galway
2022/2023

Lab
Installing (Ubuntu) Linux and First Contact with the Command Line

Semester 1

During this first lab session you will use a local Virtual Machine (VM) to install Linux on your laptop. You will use VirtualBox, which generates a virtual environment on your machine, in which you will install VMs. You will then play with some basic Linux system commands.

1 Installing Linux on your Laptop

- First, download VirtualBox and install it: `https://www.virtualbox.org/wiki/Downloads`
- Then download the latest LTS version of Ubuntu (currently 22.04.1 LTS):
  - `https://ubuntu.com/download/desktop`
- Create a Virtual Machine (VM) from the VirtualBox menu and configure it as follows:

```
- Name: ``CT213''
- Type: Linux
- Version: Ubuntu (64 bits (or 32 bits if 64 is not available)
- Memory Size: at least 2048Mb
- Hard Disk: Create a Virtual hard disk now
- Hard disk file type: VDI
- Storage on physical hard drive: Dynamically allocated
- File location: default
- File Size: at least 10Gb
```

- A VM should be created with your chosen name "CT213".
- You can start it now and it will boot.
- Point the VM to the Ubuntu installation ISO file you just downloaded.
- Choose "Try or Install Ubuntu".
- During installation, make sure you put these details:

```
- Username: ``student''
- Password: ``ct213''
- Make sure you click ``install third-party softrware''.
```

2 Using man to Get Help

Nearly every command and application in Linux will have a man (manual) file, so finding them is as simple as typing `man command` to bring up a longer manual entry for the specified command.

- For example, `man mv` will bring up the mv (move) manual.
- Move up and down the man file with the arrow keys, and quit back to the command prompt with ”q”.
- `man man` will bring up the manual entry for the man command, which is a good place to start!
- `man intro` is especially useful - it displays the ”Introduction to user commands” which is a well-written, fairly brief introduction to the Linux command line.
- Some software developers prefer info to man (for instance, GNU developers), so if you find a very widely used command or app that doesn't have a man page, it's worth checking for an info page.
- Virtually all commands understand the `-h` (or `--help`) option which will produce a short usage description of the command and its options, then exit back to the command prompt. Try `man -h` or `man --help` to see this in action.
- man pages can be lengthy, so if you are looking for a specific option etc. it could be useful to look up some word using the syntax `/word` – and then hit the key `n` to move to the next occurrence. For example, try to find the option `-t` in the man page of python.
- If you aren't sure which command or application you need to use, you can try searching the manual pages. Each manual page has a name and a short description.
  - If you know part of the command name, use the following command: `whatis -r <string>`. For example try with the following: `whatis -r cpy`
  - To search the names or descriptions for `<string>` enter: `apropos -r <string>`. For example, `apropos -r "copy files"` will list manual pages whose names or descriptions contain copy files.

3 File & Directory Commands

The Linux hierarchy (depicted in Figure 1) is typical of Unix systems (with some variations depending on the specific distributions). For the moment you just need to know that the file system is a tree that starts at the root (represented with the symbol '`/`'). A path then looks like this in Linux: `/var/log/auth.log`. This leads to the file `auth.log` in the repository `log` in the repository `var` which is right after the root of the file system.

Note that if you are familiar with DOS/Windows the path delimiter is the forward slash '`\`' and not the backward slash '`/`'.

- The tilde (˜) symbol stands for your home directory. If you are john, then the tilde (∼) stands for `/home/john`.
- `pwd`: The pwd command will allow you to know in which directory you're located (pwd stands for “print working directory”). Example: “pwd” in the Desktop directory will show “∼/Desktop”.

  Note that the GNOME Terminal also displays this information in the title bar of its window.
- `ls`: The ls command will show you ('list') the files in your current directory. Used with certain options, you can see sizes of files, when files were made, and permissions of files. Example: `ls` ∼ will show you the files that are in your home directory. Check some of the options of `ls`.
- `cd`: The cd command will allow you to change directories. When you open a terminal you will be in your home directory. To move around the file system you will use cd. Examples:
  - To navigate into the root directory, use `cd /`
  - To navigate to your home directory, use `cd` ∼
  - To navigate up one directory level, use `cd ..`

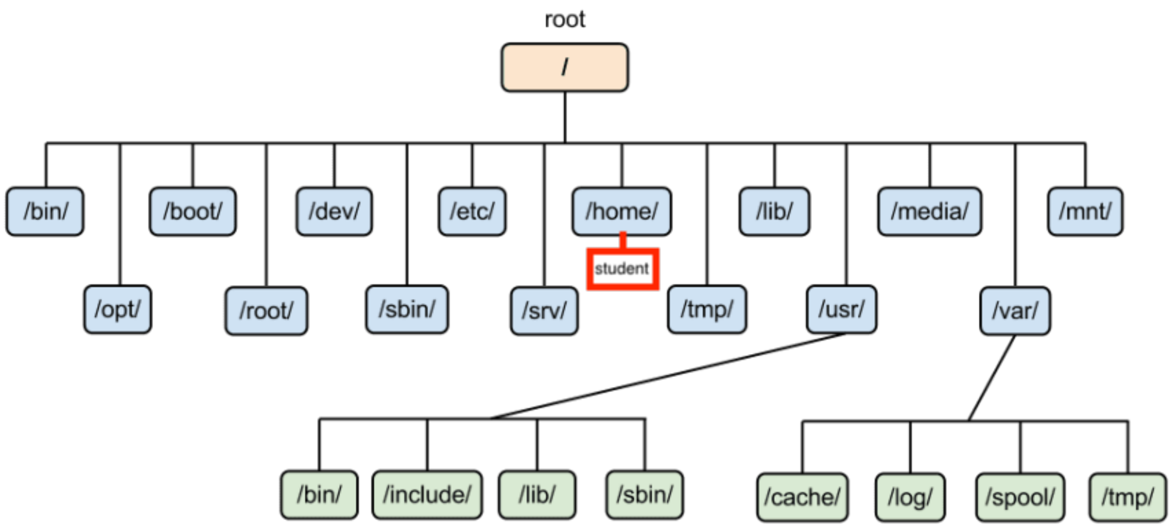


Figure 1: Linux hierarchy

- To navigate to the previous directory (or back), use `cd -`
- To navigate through multiple levels of directory at once, specify the full directory path that you want to go to. For example, use, `cd /var/log` to go directly to the `/log` subdirectory of `/var/`. As another example, `cd ∼/Desktop` will move you to the Desktop subdirectory inside your home directory.

- `cp`: The cp command will make a copy of a file for you. Example: `cp file foo` will make an exact copy of ”file” and name it “foo”, but the file “file” will still be there. If you are copying a directory, you must use `cp -r directory foo` (copy recursively). (To understand what ”recursively” means, think of it this way: to copy the directory and all its files and subdirectories and all their files and subdirectories of the subdirectories and all their files, and on and on, “recursively”)
- `mv`: The mv command will move a file to a different location or will rename a file. Examples are as follows: `mv file foo` will rename the file “file” to “foo”. `mv foo ˜/Desktop` will move the file ”foo” to your Desktop directory, but it will not rename it. You must specify a new file name to rename a file.
- `rm`: Use this command to remove or delete a file in your directory.
- `rmdir`: The rmdir command will delete an empty directory. To delete a directory and all of its contents recursively, use `rm -r` instead.
- `mkdir`: The mkdir command will allow you to create directories. Example: `mkdir music` will create a directory called ”music”.

4 Some System Information Commands

- `df`: The df command displays filesystem disk space usage for all mounted partitions. `df -h` is probably the most useful - it uses megabytes (M) and gigabytes (G) instead of blocks to report. (-h means “human-readable”)
- `du`: The du command displays the disk usage for a directory. It can either display the space used for all subdirectories or the total for the directory you run it on. Two interesting options are -s (“Summary”) and -h (“Human Readable”)
- `free`: The free command displays the amount of free and used memory in the system. `free -m` will give the information using megabytes, which is probably most useful for current computers.

- `top`: The top (‘table of processes’) command displays information on your Linux system, running processes and system resources, including CPU, RAM & swap usage and total number of tasks being run. To exit top, press “q”.
- `htop`: has my preference over `top`. You are unlikely to have it already installed so run the following command to install it:

  ```
  sudo apt-get install htop
  ```

  Play with `htop` a little to understand how it works.
- `uname -a`: The uname command with the -a option prints all system information, including machine name, kernel name & version, and a few other details. Most useful for checking which kernel you are using.
- `lsb_release -a`: The lsb_release command with the -a option prints version information for the Linux release you’re running, for example:

Lab Useful Text Related Linux Commands

Semester 1

In this second lab session you will learn other important Linux commands. Do not hesitate to check man pages and/or web pages about those commands to get more information, find more options etc.

Examining files. *Why use more if you have less*

”cat” stands for conCATenate. You can use this command to dump the entire text file to the screen.

```
$> cat /etc/passwd
```

If the text file is too long, you might find that it scrolls past too quickly and you cannot see the beginning of the file anymore. In which case, you can use either the ”more” or ”less” command.

```
$> more /etc/passwd
```

Or

```
$> less /etc/passwd
```

Both these commands perform simlar functions as they allow to see the text file one page at a time. You use the spacebar to continue paging, `<enter>` key will move down one line, and ”q” to quit. ”less” has actually more features than ”more” :) The most useful feature is that it can scroll backwards (or up) whereas ”more” cannot. Press ”h” (while in the program) to see more options. Another interesting option is the search option, which is similar to the one you’ve seen last week when we presented `man`. If you are looking for a specific string in a text file use the syntax `/string` - and then hit the key n to move to the next occurrence. For example, try to find your login name in `/etc/passwd`.

Show part of files. *Can you make heads or tails of it?*

Note: `/etc/passwd` is a file storing some login info for a user: user number, group number, shell at login, home directory etc. It also contains OS processes considered as ”users”. Use commands `whoami`, `id`, `groups` to try to get some of the information contained in `/etc/passwd`. `head` and `tail` are two opposite commands, showing the beginning or the end of a file respectively. Try the following commands – what do they give you?

```
$> head /etc/passwd
```

```
$> tail /etc/passwd
```

Both commands have various options that can be powerful – and a little complicated:

- `$> head -n 3 /etc/passwd` shows the first 3 lines
- `$> tail -n 5 /etc/passwd` shows the last 5 lines
- `$> tail -n +4 /etc/passwd` shows from the fourth line to the end
- `$> head -n -1 /etc/passwd` shows from the second line to the tenth line

A classical use of `tail` consists in showing the content of a dynamic file, that is evolving over time. `/var/log/` contains a lot of files that store log messages, i.e., information about what's happening in the system. Try `tail -f /var/log/syslog`.

Selecting Columns. *Charles I of England?*

Linux command cut is used for text processing. You can use this command to extract portion of text from a file by selecting columns.

Option -cN extracts only column (character) N from a file:

```
$> cut -c2 /etc/passwd
```

will extract only the second character of each line (the 2nd column of each line).

Range of characters can also be extracted from a file by specifying start and end position delimited with -.

```
$> cut -c2-5 /etc/passwd
```

Either start position or end position can be passed to cut command with -c option.

The following specifies only the start position before the '-'. This example extracts from 10th character to end of each line.

```
$> cut -c10- /etc/passwd
```

The following specifies only the end position after the '-'. This example extracts 10 characters from the beginning of each line.

```
$> cut -c-10 /etc/passwd
```

Instead of selecting x number of characters, if you like to extract a whole field, you can combine option -f and -d. The option -f specifies which field you want to extract, and the option -d specifies what is the field delimiter that is used in the input file.

The following example displays only first field of each lines from /etc/passwd file using the field delimiter : (colon). In this case, the 1st field is the username. The file

```
$> cut -d":" -f1 /etc/passwd
```

You can also extract more than one fields from a file. Below example displays username and home directory of users.

```
$> cut -d":" -f1,6 /etc/passwd
```

To display the range of fields specify start field and end field as shown below. In this example, we are selecting field 1 through 4, 6 and 7.

```
$> cut -d":" -f1-4,6,7 /etc/passwd
```

Sorting. *The Last Shall Be First*

This command rearranges the lines in a text file so that they are sorted lexicographically.

```
$> sort /etc/passwd
```

These are the default sorting rules:

- a number is before a letter.
- letters follow their order in the alphabet.
- a lowercase letter is before a same uppercase letter.

You can change the default sorting rules by providing a suitable parameter. E.g..

- Reverse sort:

  ```
  $> sort -r /etc/passwd
  ```

- Ignore case:

  ```
  $> sort -f /etc/passwd
  ```

You can also sort concatenate several files at once:

```
$> sort /etc/passwd /etc/group
```

You can also save the result of the sort in a another file:

```
$> sort /etc/passwd -o result.txt
```

Duplicates. *Castor Troy: It's like looking in a mirror, only not.*

This command filters duplicate adjacent lines.

```
$> uniq /etc/passwd
```

It can filter them even by ignoring the case:

```
$> uniq -i /etc/passwd
```

You can report the duplicate lines by:

```
$> uniq -d /etc/passwd
```

You can also print the occurrence of every line:

```
$> uniq -c /etc/passwd
```

Grep. *Look it up!*

Grep prints out the lines containing a string.

```
$> grep root /etc/passwd
```

Options:

- -c: only gives the number of matching lines
- -v: Shows only the lines that do not match the pattern. Inverted search.
- -i: ignore case
- -n: gives the line number as well as the matching lines.

Counting. *Things worth counting.*

Get some interesting information about the number of words, lines, characters, and bytes in the file /etc/passwd using the command wc

```
$> wc /etc/passwd
```

You can get more refined information by providing an option:

- -l: count the number of lines
- -w: count the number of words
- -c: count the number of characters

Redirecting Output. *Time to save the results.*

Common data that are generated by a program (or a command) are written into the standard output (stdout, 1). By default, the standard output stream will output text to the terminal. E.g., If you type the command "ls" on the terminal, you will get the result on the terminal. You can redirect the standard output to a file in two ways:

1. Overwrite (or creation of a new file): using ">" E.g., `ls > myfile.txt`
2. Append (or creation of a new file): using ">>" E.g., `ls >> myfile.txt`

At some point, you might also want to combine many commands: feeding the output of the first program as an input to the second one. E.g., count the number of files in a directory (`ls -l` and `wc -l`).

```
$> ls > myTmpFile.txt
$> wc -l myTmpFile.txt
```

If you do not require an intermediary file, you can use pipes '|' to redirect a stream from one program to another. E.g., you can count the number of files in a directory with one command:

```
$> ls | wc -l
$>  ls | less
```

CT213
Computing Systems & Organisation

University of Galway
2022/2023

Lab
Bash Scripts

Semester 1

First Bash Script.

To write your bash scripts you will need a text editor. There are many text editors that you can use: e.g. Nano/Vi/Vim/Emacs (if you want to edit the text directly from the terminal), Gedit (a text editor with a graphical interface on Ubuntu), or Sublime Text (a text editor with a graphical interface on Mac). You can pick any really as long as they are **text** editors and not word processors (e.g., Microsoft Word) as the latter will add some undesirable elements to the documents.

- create a new file with your text editor (convention: give it a ".sh" extension; e.g., name the file "hello.sh").
- write a simple script/program that will display "hello everyone" in the terminal (standard output – use the echo command for that). Make sure that the first line contains `#!/bin/bash`.
- make the file executable: `chmod u+x hello.sh`
- run your script (`./hello.sh`).

Arguments.

In this exercise you will play with the elements that are given as input to a script/program (by the user, from a file, from the script's arguments).

- Modify the script you implemented in the previous exercise so that it now takes an argument and is used as follows:

  ```
  $> ./hello.sh Yoda
  Hello Yoda!
  ```

- Modify your script in such a way it can now accept as many arguments as the user wants:

  ```
  $> ./hello.sh Yoda Leia Han Padme
  Hello Yoda!
  Hello Leia!
  Hello Han!
  Hello Padme!
  ```

  Or:

  ```
  $> ./hello.sh Yoda Leia "Han Solo" "Padme Amidala"
  Hello Yoda!
  Hello Leia!
  Hello Han Solo!
  Hello Padme Amidala!
  ```

You will need to use one of the structures you've seen during the lecture (for loop). This is a rough ”skeleton” of your script (a.k.a., pseudo-code):

```
#! /bin/bash
#
# for each argument from the list of arguments given as input of the script
#   print Hello followed by the current argument followed by !
```

- Modify your script: if there is no argument, your script should say hello to the Sith Lord, and in general the output should look like that:

  ```
  $> ./hello.sh Yoda
  Hello Yoda!
  $> ./hello.sh
  Hello Sith Lord!
  ```

  The pseudo-code of your script is now:

  ```
  #! /bin/bash
  #
  # if the script has no argument then
  #   print Hello followed by "Sith Lord" followed by !
  # else
  #   for each argument from the list of arguments given as input of the script
  #     print Hello followed by the current argument followed by !
  ```

- Now I want your script to be... rude. Make sure it says hello only to every second character:

  ```
  $> ./hello.sh Yoda Leia "Han Solo" "Padme Amidala"
  Hello Leia!
  Hello Padme Amidala!
  ```

  ```
  #! /bin/bash
  #
  # if the script has no argument then
  #   print Hello followed by "Sith Lord" followed by !
  # else
  #   while the number of arguments is greater than 0
  #     print Hello followed by the current argument followed by !
  #     shift the arguments (command shift)
  #     shift the arguments (command shift)
  #
  ```

Echo and Special Characters.

The goal of this section is to play with special characters and to show how they impact the execution of commands, and can be disabled conveniently. If you need to interrupt/stop a command, you can use the combination control + c. Many of the little exercises in this section look similar to what you've seen in the lecture so you should not hesitate to just copy and try the examples from the lectures.

- print on the screen a simple word, for instance hello (use echo).
- echo can print several strings - print ”hello everybody”
- you can also use echo to print non alphabetical characters on screen (as long as they do not belong to the list of special characters). Use echo to print ”Hello to the 2 of you!!”

- Disable the special character in the following string (using the character \) and print it on screen using echo: ”# is a very useful character in bash”
- Likewise for the following: ”# is more useful than \in bash”
- Print the following using echo: ”# is more useful than \in bash but less than *”
- When you need to disable a lot of special characters, you can use simple and double quotes to disable all (simple quotes) or some (double quotes - do not disable $, ‘ and \) of them. Print the following two sentences using simple then double quotes: ”# is less useful than * in bash” and ”# is more useful than \in bash”

CT213
Computing Systems & Organisation

University of Galway
2022/2023

Lab

Review of the Concepts Introduced so Far

Semester 1

1 A Simple Counter.

The goal of this week's exercise is to write a script that increments a number contained in a file a every second.

- The script first checks if the arguments (given to the script) are correct: i.e., (i) there is one and only one argument and (ii) the argument is a string that describes an existing directory (condition to use: `[ -d "path" ]`) followed by a file name. Use the commands `basename` and `dirname` (check their man pages) to get the directory and the file in the path given as argument of the script. Write a script that returns 1 (exit code for "`false`" or "`not ok`") if the arguments are invalid or else returns 0 (exit code for "`true`" or "`ok`"). If the arguments are invalid, the scripts should also print a message on the standard error output (typically: the screen).

  ```
  $> ./counter.sh /home/takfarinas/CT213/counter.txt
  The directory ``/home/takfarinas/CT213/'' is invalid
  $> echo $?
  1
  $> ./counter.sh /home/takfarinas/CT213//counter.txt
  $> echo $?
  0
  ```

- If the file given as last element of the path argument does not exist yet - create it and write "`0`" in the file.
- Now your script will run an endless loop and, each second, will increment the number contained in the file. The different "steps" or "instructions" that you need to implement are: (i) read the number in the file and assign it to a variable (ii) add 1 to the number (using command `expr`) (iii) write the new number in the file (iv) wait 1 second (command `sleep` which just waits for a certain period given as argument). An endless loop looks like: `while true; do ... done`. To stop a script that does not seem to terminate or is running and endless loop, use the combination of keys `control + c`.

While the problem described in this exercise was easy to understand (a file contains a simple integer and every second your script has to increment the number in the file) the steps required to get to a working program (script) are quite complex. In this exercise, I will introduce each of these steps in more details and you will do the integration yourselves. The next sections will describe, one after the other, one element of the final script each.

1.1 expr

At some point in your script, you will have to manipulate arithmetic expressions. In bash, the command that you should use to play with arithmetic expressions is `expr`. Try the following in the command line:

```
$> expr 1 + 1
2
$> myvar=$(expr 1 + 1)
$> echo $myvar
2
```

```
$> expr $myvar + 1
3
$> expr $myvar / 3
0
$> expr $myvar \* 3
9
Etc.
```

Check `expr`'s man page and test other arithmetic expressions.

1.2 Basename and Dirname

In your script you will have to evaluate a path (e.g., `/home/takfarinas/CT213/my_file.txt`) in order to extract some info regarding the directory (and file) contained in the path. In the example given the directory is `/home/takfarinas/CT213` and the file `my_file.txt`. To obtain these two elements, use the following commands:

```
$> dirname /home/takfarinas/CT213/my_file.txt
/home/takfarinas/CT213
$> basename /home/takfarinas/CT213/my_file.txt
my_file.txt
```

Play a little bit with these commands and check the man pages in any doubts.

1.3 Redirection of Standard I/O Streams

In Bash, when a program is executed, it uses three standard I/O streams numbered 0, 1, and 2

- 0: `stdin`, the standard input stream
- 1: `stdout`, the standard output stream
- 2: `stderr`, the standard error stream

These standard I/O streams can be redirected using the `X>` operator, where `X` is the file descriptor number.

For instance, by default, the `stdout` stream (i.e., 1) will output text to the terminal. However, you can redirect it to a file using ">" (e.g., `ls > result.txt`) or "1>" (e.g., `ls 1> result.txt`).

However, sometimes, you would like to redirect the `stderr` (i.e., 2) stream if you wish to store the error message of some command in a file. You can do that using the "2>" (e.g., `ls inexistingFile.txt 2> result.txt`). You can also use this to suppress error messages from being displayed by redirecting the `stderr` to `/dev/null` (e.g., `ls 2> /dev/null`).

Conversely, you might want to save both the standard output and the errors of a program in the same file. Therefore, you need to redirect for the redirect the `stdout` to your result file, but also redirect the `stderr` into the `stdout` (e.g., `ls > result.txt 2>&1`).

Also, you might want you program to show an error, but you would like to make sure that you do not mix between regular output results and errors. Therefore, you need to redirect the error message to `stderr` (e.g., `echo "Something is wrong" >&2`).

1.4 Checking the arguments of a script

Commands, scripts, or a programs often require a certain number or arguments - not more, or less, than a certain number. This is also good practice to check the number of arguments in your scripts to avoid (or track) certain ty As a firstrs. In this section I will give you a sort of header that you can use in all your scripts, checking the number of parameters given. You can reuse and adapt for all your scripts from now on. pes of erro step, implement the following skeleton in a bash file

```
#!/bin/bash

# this is an example of how to check the arguments
if [ $# -ne 1 ]; then
```

```
  echo "the number of parameters is wrong" >&2 # &2 is standard error output
  exit 1 # the exit code that shows something wrong happened
fi

# here the core of your script

# at the end of the script an exit code 0 means everything went well
exit 0
```

This skeleton, or pattern, will be used in many of the scripts you'll write.

1.5 Command Substitution

Command substitution reassigns the output of a command; it literally plugs the command output into another context. The classic form of command substitution uses backquotes (`` `...` ``). Commands within backquotes (backticks) generate command-line text.

A simple example is: `` ls `pwd` `` that gives the output of `pwd` to `ls`.

An example in a script:

```
#! /bin/bash

script_name=`basename $0`
echo "The name of this script is $script_name."
```

1.6 Checking Whether a Directory or a File Exist

In our script we need to know:

- whether a directory exists
- whether a file in a directory exists

The two conditions that we want to check are the following: `[ -d path ]` to test if path corresponds to a valid directory and `[ -f path ]` which is true if path leads to a regular file.

Other file test operators: `http://www.tldp.org/LDP/abs/html/fto.html`

Example corresponding to the tests that you need to perform for your script:

```
#! /bin/bash

filename=$1
dir_name=`dirname $filename`
base_name=`basename $filename`

# this is an example of how to check the arguments
if [ $# -ne 1 ]; then
  echo "the number of parameters is wrong" >&2 # &2 is standard error output
  exit 1 # the exit code that shows something wrong happened
fi

#check whether the argument leads to a directory...
if ! [ -d $dir_name ]; then
  echo "the directory in $dir_name is wrong" >&2 # &2 is standard error output
  exit 2 # the exit code that shows something wrong happened with the directory
fi
#... and a file
if ! [ -f $base_name ]; then
  echo "the file in $base_name is wrong" >&2 # &2 is standard error output
  exit 3 # the exit code that shows something wrong happened with the file
fi
```

```
# here the core of your script

# at the end of the script an exit code 0 means everything went well
exit 0
```

Write the script and test it with a few examples to see whether it works.

1.7 Read from a File

Depending on what the file contains and how you want to exploit it, you will have to use a different method. The most common cases are:

- Reading the content of the file into a variable:

  ```
  $> myVariable=`cat myFilename`
  ```

  Notice that if the value you are reading from the file is an integer and you would like to do some operations with the variable (e.g., addition, multiplication, etc.), you would need to explicitly indicate the type:

  ```
  $> typeset -i myIntVariable=`cat myFilename`
  ```

- Reading the content word by word:

  ```
  $> myVariable=`cat myFilename`
  $> for word in $myVariable; do
  .... echo "word: $word"
  .... done
  ```

- To read the file line by line, you would need to modify the IFS (Internal Field Separator) from space ($' ') to newline ($ '\n'):

  ```
  $> IFS=$'\n'
  $> myVariable=`cat myFilename`
  $> for line in $myVariable; do
  .... echo "line: $line"
  .... done
  ```

1.8 Write in a File

Similarly to reading from a file, writing in a file depends on the writing mode (append/overwrite) and on the data you would like to write in (one/many lines):

- You can redirect the standard output of any program/command into a file using > for overwriting or >> for appending to the end of the file:

  ```
  $> echo "Hello World!" > myFilename.txt
  $> echo "Hello World!" >> myFilename.txt
  ```

  Note that writing in a file that does not exist leads to the creation of new a file with the same name. However, an error is returned if the base directory of the file does not exist.

- You can write a single line in the file using "echo" :

  ```
  $> echo "Hello World!" >> myFilename.txt
  ```

  You can also write a line in the file without adding a newline at its end:

  ```
  $> echo -n "Hello World!" >> myFilename.txt
  ```

  Similarly, you can also write two or more lines at once using the newline delimiter `\n`:

  ```
  $> echo -e "First line\nSecond Line" >> myFilename.txt
  ```

1.9 Our Script

Now you should be able to write the script/program corresponding to the Simple Counter problem described at the start of this lab!

CT213
Computing Systems & Organisation

University of Galway
2022/2023

Lab
Synchronisation (Semaphores)

Semester 1

1 Synchronisation Issues

1.1 Symbolic Links

A symbolic link (also known as soft link), is a special kind of file that points to another file. This is similar to a shortcut in Windows or a Macintosh alias. You can create a symbolic link to a file using the command `ln` (e.g., `ln -s file1.txt linkFile1.txt`).

Unlike a hard link, a symbolic link does not contain the data in the target file. It simply points to another entry somewhere in the file system.

1. Run the command `ln -s "$0" "$1"` in the terminal, with `"$0"` being any file (this file should exist) and `"$1"` being any random name (not the name of an existing file). What do you see?

2. Now, write the following script `test_ln.sh` and check what happens if `"$1"` is an existing file (Answer: the loop should never finish and you don't have the prompt any more). Don't stop the script now.

```bash
#! /bin/bash

# the 'link' (ln) system call is supposed to be atomic on a local file system
while ! ln -s "$0" "$1" 2> /dev/null; do
    sleep 1
done
```

Listing 1: test_ln.sh

4. Now open another terminal (or another tab in the same terminal) and remove the file (symbolic link) that was used as `"$1"` in question 2. What do you observe?

5. What does the commented line in the script says? Why does the atomicity of the `ln` command matter?

1.2 Symbolic Links as Locks

The atomicity guaranties that both commands: (i) checking whether a link with the specified name exists and (ii) the creation of a new link with that name cannot be interleaved (e.g., another program cannot create link with that same name in between). Therefore, we could use the existence/absence of a symbolic link with the given name as a locking mechanism as follows:

- If the link already exists, then the lock is taken by another process and we have to wait until the link no longer exists, then this process will create the link (i.e., acquire the lock). See script `acquire.sh`.

- The process holding the lock deletes the link it created when it wants to release the lock. See script `release.sh`.

```
#! /bin/bash

if [ -z "$1" ]; then
    echo "Usage $0 mutex-name" >&1
    exit 1
else
    # the 'link' (ln) system call is supposed to be atomic on a local file system
    while ! ln "$0" "$1" 2>/dev/null; do
        sleep 1
    done

    exit 0
fi
```

Listing 2: acquire.sh

```
#! /bin/bash

if [ -z "$1" ]; then
    echo "Usage $0 mutex-name" >&1
    exit 1
else
    rm "$1"
    exit 0
fi
```

Listing 3: release.sh

1.3 Synchronisation Issue

Here is the code for the script `write.sh`:

```
#! /bin/bash

if [ $# -lt 1 ] ; then
  echo "This script requires at least one parameter"
  exit 1
fi
for elem in "$@" ; do
  if [ ! -e "$elem" ] ; then
    echo 1st $$ > $elem
  else
    echo next $$ >> $elem
  fi
done
```

Listing 4: write.sh

1. run the following command twice (or more): `./write.sh a b c` and explain the content of the files. Read the files using less (for instance. the Variable `$$` corresponds to the PID of the process running a script.

2. Now run the script `run_write.sh` which exhibits synchronisation problems (two processes accessing files concurrently). Check how many lines are there in the files (there should be two but because of synchronisation issues there may be only one in some of the files)? You can check the number of lines using `wc -l file_name`: `wc -l f1 f2 f3` in our case. Try several times if you see 2 lines in each files – until you see the problem. Why does the problem occur?

```
#! /bin/bash

rm -f f1 f2 f3

./write.sh f1 f2 f3 & ./write.sh f1 f2 f3
```

Listing 5: run_write.sh

4. Identify the *Critical Section* in `write.sh`

5. Use the two scripts `acquire.sh` and `release.sh` given above and modify `write.sh` to ensure mutual exclusion on the critical section.
6. Make sure that the solution you proposed does not block processes which do not access the same file.

CT213
Computing Systems & Organisation

University of Galway
2022/2023

Lab
Inter Process Communitation (IPC) 1: Named Pipes

Semester 1

This practical will introduce named pipes as a way for processes to communicate. As an introduction you can probably have a look at the following (old) articles (from the Linux Journal):

- `http://www.linuxjournal.com/content/using-named-pipes-fifos-bash`
- `http://www.linuxjournal.com/article/2156`

You'll obviously find plenty of other resources online...

1 A Ring of Communicating Processes

In this exercise we want to create several (similar) processes and make them communicate (communication between them is one-way in this exercise). The general architecture of our system can be seen in Figure 1 with 6 processes and 6 named pipes between them. Note the order (names) of the pipes and the processes: `pipe1` is between processes `p1` and `p2` etc. Actually what we want is for messages from `p1` to `p2` (and only those ones) to use `pipe1` – there will not be any symmetric message between `p2` and `p1` in our exercise. This means that if `p1` can send messages to `p2`, `p2` cannot send anything to `p1` and vice-versa, and so on for every couple of processes.

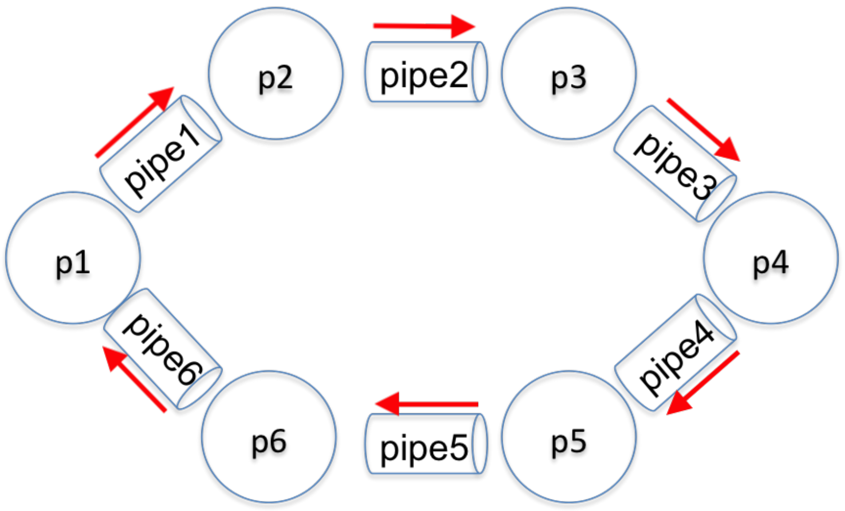


Figure 1: Architecture of our System

1.1 Simple Test

Create a named pipe called `test_pipe` (using the command `mkfifo test_pipe`). As a first example we write/read one message to/from the named pipe.

The commands `echo` and `read` should be used to write and read to/from a named pipe. Use **two** terminals to test your scripts.

- Terminal 1, write using the command: `echo "something" > test_pipe`
- Terminal 2, read using the command: `read input < test_pipe; echo $input`

1.2 Test with Loops

Now create two scripts `write_sh` and `read_sh` so that:

- `write_sh`: the writer continuously writes in the pipe something that is given by the user on Terminal 1
- `read_sh`: the reader continuously reads from the pipe and displays it on Terminal 2.

The loop for the writer should be:

```
#!/bin/bash

while true; do
        read input
        echo $input > test_pipe
done
```

Listing 1: write.sh

The loop for the reader should be:

```
#!/bin/bash

while true; do
        read input < test_pipe
        echo received from the pipe: $input
done
```

Listing 2: read.sh

Again, use two terminals to test your scripts. You can stop both scripts with `control+c`.

1.3 In and Out

Now write one additional script (name it `inout.sh`) that does one thing: read from one named pipe and write in another named pipe. this script should have two parameters (the two named pipes). Use an endless loop for this and reuse the two scripts `write_test.sh` and `read_test.sh` that you have implemented before (remember: read THEN write).

Note that you have **2 DIFFERENT** pipes here, one for the input and one for the output (in the scripts: `write_test.sh` and `read_test.sh` the pipe was the same).

Create two named pipes `pipe_test1` and `pipe_test2`, and test your script: write in the “in” named pipe and check what is in the “out” pipe (from another terminal using `read input < pipe; echo $input}`.

The following is a STEP in the right direction (things are missing):

```
#!/bin/bash

in_pipe=$1
out_pipe=$2

while true; do
        read input < XXX
        echo I found this in the pipe: $input and I am going to send it on my out pipe
        echo $input > YYY
done
```

Listing 3: inout.sh

1.4 First Attempt

Now create 6 named pipes and start 6 of your `inout.sh` processes with the correct parameters (previous and next pipes)–each process in its own terminal. Then, open another terminal and start the writing process: write something in one pipe and observe the processes sending the input to each other. Stop the processes with control+c. You probably realise there is something wrong here.

1.5 Stop!

Now we want to stop the input to be sent again and again and again. When a script reads the same word twice in a row, it does not forward it on to its neighbour. This would stop the problem you've seen before. Test that the processes are not in an endless loop anymore, sending the same message for ever.

The following is a STEP in the right direction (things are missing):

```
#!/bin/bash

in_pipe=$1
out_pipe=$2

prev_input="init"

while true; do
        read input < XXX
        echo I found this in the pipe: $input and I am going to send it on my out pipe
        if [ YYY ]; then
                echo $input > ZZZ
                prev_input=$input
        fi
done
```

Listing 4: inout.sh

1.5.1 A Better Solution *(Optional)*

What if we don't want to initialise `prev_input`? What other condition/if/test do you need to add?

1.6 A Nice Solution *(Optional)*

Obviously the problem at the moment is that you need to know the name of the two named pipes (in and out) for each process. Ideally, the scripts should create a named pipe at the start of the script (say the in named pipe), then ask for the other one (say, the out one). To do so, they should communicate with another script, let's call it a bootstrap process, which knows who's who and can tell every process which pipe they should write to. The idea is simple: at the start of every "normal" process, the process creates a named pipe and sends a message to the bootstrap process which stores the name of the named pipe and writes in it the other named pipe.

```
#!/bin/bash

read prev_pipe < pipeb
first_pipe=$prev_pipe

for var in 1 2 3 4 5; do
        read next_pipe < pipeb
        echo in pipe: $next_pipe > $prev_pipe
        echo sent $next_pipe in $prev_pipe
        prev_pipe=$next_pipe
done
echo in pipe: $next_pipe > $first_pipe
echo finally send $next_pipe in $first_pipe
```

Listing 5: bootstrap.sh

Modify your previous "repeater" script to (i) create a named pipe (`mkfifo`) and then send the name of the named pipe on the boostrap process' named pipe (ii) read on the pipe the name of the out pipe. Launch the bootstrap process. Launch the repeater processes and check they get the correct named pipes.

Test that the ring is connected.

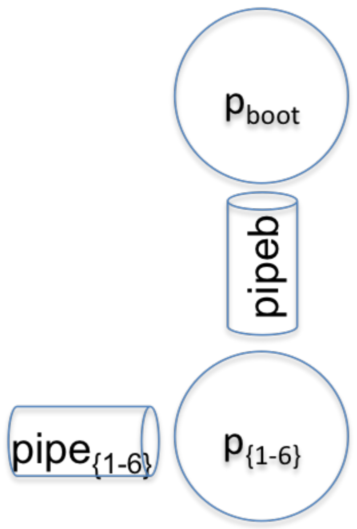


Figure 2: Architecture of the Bootstrap Mechanism