School of Computer Science

COMP30640

Lab 5
Synchronisation I

Teaching Assistant:	Thomas Laurent
Coordinator:	Anthony Ventresque
Date:	Friday 12th October, 2018
Total Number of Pages:	7

1 Exit code.

On Linux systems, programs can pass a value to their parent process while terminating. This value is referred to as an exit code or exit status. The standard convention is for the program to pass 0 for successful executions and 1 or higher for failed executions. You can access the exit code of your most recent command using the variable `$?`, for instance in the command line interpreter. Try the following sequence of commands in your terminal:

```
$> touch test1
$> echo $?
0
$> touchtest2
touchtest2: command not found...
$> echo $?
127
$> touch /etc/test3
touch: cannot touch /etc/test3: Permission denied
$> echo $?
1
```

You can see a series of 3 commands (`touch test1`, `touchtest2` and `touch /etc/test3`), each of them followed by the command `echo $?`. The first command `touch test1` creates a file "test1" in your current directory (touch is a command that creates new files if they do not exist). I assume the command will be successful and the exit code is **0**. The second command `touchtest2` has a typo and does not work (see the error message). In this case, the exit code is **127** (it's a specific code for a command which is unknown from the system). The third command `touch /etc/test3` tries to create a new file in a directory you cannot modify and the system tells you don't have the correct permission for it. The exit code here is **1**.

Exit codes will be useful in your scripts as you can use them to:

- check that a script has terminated correctly
- test the results of some commands (e.g., in if statements or while/for loops)

2 Checking the number of arguments to a script.

Programs in Bash often require a certain number or arguments - either at least a number, or exactly a number, or not more than a number. It is then good practice to begin your scripts with a block of instructions that checks the number of arguments. Look at the following program skeleton: it contains a header that you can use in all your scripts from now on, checking the number of parameters given. You should be able to understand everything in this skeleton/script - everything should be in the slides of the special deck on Bash and previous weeks' practicals. Note that # is used to comment what's on the right of the symbol: a whole line or part of a line.

```
#!/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 from now on.

3 Ampersand. &

A single ampersand & can often be found at the end of a command in Bash. It is used to send the command in the **background** (see last week's practical). In particular, it can be used to do multiprogramming: creating various processes in parallel. Create the little script below:

```
#!/bin/bash

for var in 0 1 2 3; do
        echo $1 $var
        sleep 1
done
```

Now run the following series of commands and explain what you see:

```
$> ./test_loop.sh A
A 0
A 1
A 2
A 3
$> ./test_loop.sh A &
[1] 26856
$> A 0
A 1
A 2
A 3
#type enter
[1]+  Done                    ./test_loop.sh A
$> ./test_loop.sh A ; ./test_loop.sh B
A 0
A 1
A 2
```

```
A 3
B 0
B 1
B 2
B 3
$> ./test_loop.sh A & ./test_loop.sh B &
[1] 26911
[2] 26912
$> B 0
A 0
B 1
A 1
B 2
A 2
B 3
A 3
#type enter
[1]-  Done                    ./test_loop.sh A
[2]+  Done                    ./test_loop.sh B
```

Note that there is a slight bug/inconvenience in this script. Although the '&' detaches STDIN, it does not do the same for STDOUT and STDERR, meaning that these are still visible in the console. You can direct STDOUT to a file or /dev/null to fix this issue, as shown below.

```
#!/bin/bash

for var in 0 1 2 3; do
        echo $1 $var >/dev/null
        sleep 1
done
```

4 Basename and dirname

Your scripts will sometime have to evaluate a path (e.g., `/home/anthony/COMP30640/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/anthony/COMP30640` and the file `my_file.txt`. To obtain these two elements, use the following commands:

```
$> dirname /home/anthony/COMP30640/my_file.txt
/home/anthony/COMP30640
$> basename /home/anthony/COMP30640/my_file.txt
my_file.txt
```

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

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."
```

As an exercise, you can try to create a command that automatically kills a specific process – look at last week's practical. First run on of the `yes` commands we've played with last week and then try to identify the process number using `ps`, `grep`, `cut` (find the line containing yes, then split the line on the specific field related to the PID). Use then `kill -9 ‘the-command-you-just-created‘`.

Solution

```
$> kill `ps | grep yes | cut -d" " -f2`
```

5.1 Checking Whether a Directory or a File Exist

In some scripts we will 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 $dir_name/$base_name ]; then #£dir_name/£base_name is the same as £1
  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.

6 Synchronisation Issues.

1. Create a script called `write.sh` containing the code given below. Run the following command twice (or more): `./write.sh a b c` and check the content of the files. Explain the content of the generated files. Note that the variable `$$` corresponds to the PID of the process running a script. Check that using the **ps** command (cf. last week's practical).

   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
   ```

2. Now create and run the script `run_write.sh` (code below) which exhibits synchronisation problems (two processes accessing files concurrently). Check how many lines there are 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?

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

```
#! /bin/bash

rm -f f1 f2 f3

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

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

4. Now lets look at some of the solutions from the lecture that didn't work so well in action. Take a look at the pseudo code from solutions 2 and 3 from the lecture slides (Lecture 7: slides 10 and 11) and implement these in bash.

   - use 'touch' to create 'note' and 'milk' files.
   - to check if a file exists use '-f' in the conditional statement.
   - to check if a file doesn't exist, use '! -f' in the conditional.
   - to mimic starvation (i.e. B going on holidays), you can play with the 'sleep' command ('sleep 20' means that the program will wait for 20 seconds).

5. Run each script on its own and then in parallel (./processA.sh & ./processB.sh) What do you notice? Why don't they work?

Solution 2

```
# processA_2.sh
if [ ! -f 'note' ]; then
        if [ ! -f 'milk' ]; then
                touch "milk"
        fi
    touch note
fi

# processB_2.sh

if [ -f 'note' ]; then
    if [ ! -f 'milk' ]; then
        touch "milk"
        #sleep 20
    fi
    rm note
fi
```

If we put a note in place and let processB sleep for a few seconds, we can see the starvation problem. processA is left waiting on processB.

Solution 3

```
# processA_3.sh
touch note_A

if [ ! -f note_B  ]; then
        if [ ! -f milk ]; then
                touch milk
    fi
fi
rm note_A

# processB_3.sh

touch note_B

if [ ! -f note_A  ]; then
        if [ ! -f milk ]; then
                touch milk
        fi
fi
rm note_B
```

If both put notes at the same time, then both won't enter the if statement meaning neither will buy milk.