Exercise Sheet 4

The dark side of Bash

29 October 2020

Exercise 1

Two concurrent processes

In this exercise we will assume that we can trust the operating system kernel and, in particular, we will use bash to handle concurrent processes, although we learnt this morning that, strictly speaking, we do not have any guarantee that a PID will not be reused by a different process than that we have in mind.

Suppose to have a simulation that runs for a very long time and that you have to perform some operations from time to time to monitor it. In your script, you can use the following two functions to mock the two tasks.

```
function Simulate()
{
    local index
    for((index=0; index<10; index++)); do
        echo "In ${FUNCNAME}: index=${index}"
        # ((index==7)) && return 1
        sleep 3
    done
}

function Monitor()
{
    local simulationPid sleepTime
    simulationPid="$1"
    sleepTime="$2"
    while kill -0 "${simulationPid}" 2>/dev/null; do
        echo "Simulation still running, sleeping ${sleepTime}s..."
        sleep "${sleepTime}"
        # return 2
    done
}
```

How would you let them start at the same time in a way that the `Monitor` function is really *monitoring* the `Simulate` one? Implement a mechanism to detect possible failures of either task. In particular,

- if `Simulate` fails, you should wait that `Monitor` finishes, report the error and exit;
- if `Monitor` fails, you should make `Simulate` abort, report the error and exit;
- if some other relevant case can happen, be sure to cover it somehow in your script.

Uncomment and/or modify the commented-out lines to mimic failures.

Exercise 2

Using traps to clean up

The main purpose of this exercise is to understand how a trap works and how it can be useful. We will consider the most common case, cleaning up, but signal handling can be used for much more than that.

Suppose you have an analysis tool that takes a file with only one column in input and produces a file in output. Suppose as well that your tool produces temporary files which are auxiliary for the analysis. We will mimic the analysis tool via `sort -g` (redirecting the output) and the temporary

files will be simply touched. Write a script to complete the following operations.

1. `cut` the data file into several files, each having one column of the original file.
2. For each single-column file
   - process it with the analysis tool (which will produce an output file);
   - create the auxiliary files, e.g. with `touch column_${index}_{0..9}.aux` (or similar);
   - sleep for one second;
   - remove the auxiliary files.

You can produce a fake data file via `seq 0 .001 1 | shuf | columns -c 11 -W 100` which has 11 columns, each with 91 lines.

3. Run the script (ideally in a new folder) and abort it with CTRL-C after few seconds. Your folder should have been polluted by a bunch of files.
4. Add a `trap` to your script to clean up on exit.

Exercise 3

Exploring bash behaviour with `errexit` options enabled

You have learnt today that the `errexit` shell option of bash requires some effort in order to understand how it works. Here you have the opportunity to deepen in. Using the slides or the bash manual as guideline, try to predict the output of the following scripts. The last three have been taken from the Greg's Wiki and you should try to understand why they behave differently.

1.
```bash
#!/usr/bin/env bash
( set -e; ( echo 11; false; echo 22 ) ; echo 33 ); echo 44
```
2.
```bash
#!/usr/bin/env bash
function F1() ( set -e ; F2 ; F3 ) # Subshell!
function F2()  { echo 'In F2'; false ; }
function F3()  { echo 'In F3'; false ; }

echo 'Call: F1';              F1
echo 'Call: F1 || false';     F1 || false
```
3.
```bash
#!/usr/bin/env bash
set -e
[[ -d notExistingDir ]] && echo 'Dir found'
echo 'Survived'
```
4.
```bash
#!/usr/bin/env bash
set -e
function F1() { [[ -d notExistingDir ]] && echo 'Dir found'; }
F1
echo 'Survived'
```
5.
```bash
#!/usr/bin/env bash
set -e
function F1() { if [[ -d notExistingDir ]]; then echo 'Dir found'; fi; }
F1
echo 'Survived'
```

Exercise 4

Some tasks for which Sed can provide good support

Go through the few slides discussed in the lecture about `sed`, focusing on the examples there. Once prepared an input file via

```
paste <(shuf -n 20 /usr/share/dict/ngerman) <(shuf -i 1-5000 -n 20)
```

achieve the following tasks using a `sed` command. You can assume that the file contains a word in the first column and an integer number in the second one.

1. Print the first 5 and last 10 lines.
2. Print every third line.
3. Print lines starting by a vowel.
4. Print lines for which the integer number is larger than 999.
5. Print the number of lines in the file.
6. Print lines for which the number in the second column is multiple of 5.