Scientific Programming – Concurrent programming in Python

Exercises:

1) Have a closer look at the first example of multithreading in Python, shown at the end of the first section of the lecture (slide 15). You will find the file (`02-1-simple_threads.py`) also on the course's WeBeep page (in materials → exercises).

Run the program (preferably with Python 3) and observe what happens. Run it several times! Do you always get the same output? If you do, can you explain why? And if you don't, can you explain why not?

2) Modify the program by commenting out the two join lines close to the end. [`thread1.join()` and `thread2.join()`]

Run the program again and observe the difference. Can you explain why this occurs?

3) Add a third thread (same delay) and run the program several times to observe the differences.

4) So far, each thread has an own counter, so there cannot be a race condition. Let's see what happens if the threads all work on the same counter …

For this purpose, add the following line at the top, after the import-statements:

```
counter = 15
```

This defines a global variable for the entire program.

Now, within `def thread_count_down…`, replace

```
counter = 5
```

by

```
global counter
```

This means, that within that function, instead of defining a local variable `counter`, the global variable will be accessed.

Run the program again and try to figure out what happens … can you explain all the strange behavior???

5) Implement the queue example (on slide 32) from the lectures and run it.

- In what order are the tasks present in the "Tasks" queue?

- In what order are the results stored in the “Results” queue?
- What can you tell about the method `Queue.get`? What does it do?

6) Reimplement the `simple_threads.py` script, used for the last exercise, using multiprocessing instead of threading (create the corresponding `simple_multiprocessing.py`)!

a) Run the same tests described in the lecture (see slide 19), that helped us to determine that mutithreading in Python doesn’t behave as expected from many other programming languages:
   - Run 2 and 4 processes sequentially
   - Run 2 and 4 processes in parallel
   - Run 2 and 4 processes in parallel, computing the running sum of 1 to 100 million instead of waiting for 0.5 seconds

b) Verify the run times of the tests and compare them to those obtained for multithreading.

c) Pay especially attention to the runtimes obtained for the running sum of 1 to 100 million: do you obtain largely different runtimes for 1, 2 and 4 processes (like it was the case for multithreading)? Do you think the processes run in parallel?

d) Pay attention to the final sum you obtain from the processes when computing the running sum. Do they differ for the different processes or runs? If so, why? If not, why not?

e) Let the main program and the spawned processes additionally print out the system process ID and verify that they are indeed all different!

7) Pooling: take the example from the lectures and test what happens

- if the number of processes in the pool is lower than the number of data points to compute
- if the number of processes in the pool is higher than the number of data points to compute [Hint: observe the process IDs.]

8) Implement a deposit/withdrawal program which has two bank accounts in shared memory and spawns four processes:

- Process A: 10,000 transactions to transfer a small amount (1) from account 1 to account 2
- Process B: 10,000 transactions to transfer a small amount (1) from account 2 to account 1
- Process C: 10,000 deposits of a small amount (1) to both accounts
- Process D: 10,000 withdrawals of a small amount (1) from both accounts

Your appropriate locks to make sure that after all processes have finished, both bank accounts have the same balance they had in the beginning.
Would you get the same result if you didn’t use locks? Give it a try, if you want!

Hint: if your program doesn’t print anything, then maybe you have a deadlock? ... :-)