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?

At first glance, the output might appear to be always the same, but it is not. The two threads are not precisely synchronized and sometimes thread A is the first to finish while sometimes thread B finishes earlier:

```
$ python3 02-1-simple_threads.py
Starting thread A.
Starting thread B.
Thread A counting down: 5...
Thread B counting down: 5...
Thread A counting down: 4...
Thread B counting down: 4...
Thread A counting down: 3...
Thread B counting down: 3...
Thread A counting down: 2...
Thread B counting down: 2...
Thread A counting down: 1...
Finished thread A.
Thread B counting down: 1...
Finished thread B.
Finished.

$ python3 02-1-simple_threads.py
Starting thread A.
Starting thread B.
Thread A counting down: 5...
Thread B counting down: 5...
Thread A counting down: 4...
Thread B counting down: 4...
Thread A counting down: 3...
Thread B counting down: 3...
Thread A counting down: 2...
Thread B counting down: 2...
Thread B counting down: 1...
Finished thread B.
Thread A counting down: 1...
Finished thread A.
Finished.
```

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?

```
$ python3 02-1-simple_threads-modified.py
Starting thread A.
Starting thread B.
Finished.
Thread A counting down: 5...
Thread B counting down: 5...
Thread A counting down: 4...
Thread B counting down: 4...
Thread A counting down: 3...
Thread B counting down: 3...
Thread A counting down: 2...
Thread B counting down: 2...
Thread A counting down: 1...
Finished thread A.
Thread B counting down: 1...
Finished thread B.
```

Why does this occur? The main program creates two threads A and B and continues with it's own code reaching the last comman print ('Finished.') before the new treads start performing their tasks. With the join() commands the main program waits for the threads to finish before proceeding with it's own code.

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???

```
$ python3 02-1-simple_threads-modified.py
Starting thread A.
Starting thread B.
Thread A counting down: 15...
Thread B counting down: 14...
Thread B counting down: 13...
Thread A counting down: 13...
Thread A counting down: 11...
Thread B counting down: 10...
Thread B counting down: 9...
Thread A counting down: 8...
Thread B counting down: 7...
Thread A counting down: 7...
Thread B counting down: 5...
Thread A counting down: 4...
Thread B counting down: 3...
Thread A counting down: 2...
Thread B counting down: 1...
Finished thread B.
Thread A counting down: 0...
Thread A counting down: -1...
Thread A counting down: -2...
Thread A counting down: -3...
Thread A counting down: -4...
Thread A counting down: -5...
Thread A counting down: -6...
Thread A counting down: -7...
[...]
```

We have to interrupt the program otherwise it continues to count downwards… Both threads access the same counter, when one of them sets it to 0, the next to call to while counter in thread_count_down() in one of the two threads will cause that thread to finish, but very likely the other thread then subtracts 1 and while -1 does not quit, so the second thread continues to count down infinitely …

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?

```
$ python3 source-code/02-2-queue.py
Process-1 : 0 * 2 = 0
Process-2 : 1 * 2 = 2
Process-3 : 2 * 2 = 4
Process-4 : 3 * 2 = 6
```

```
Process-6 : 5 * 2 = 10
Process-7 : 6 * 2 = 12
Process-8 : 7 * 2 = 14
Process-9 : 8 * 2 = 16
Process-5 : 4 * 2 = 8
Process-10 : 9 * 2 = 18
[…]
Process-100 : 99 * 2 = 198
Process-97 : 96 * 2 = 192
Process-90 : 89 * 2 = 178
```

Although the tasks are stored in numeric order in the “Tasks” queue (`for t in range(n): myTasks.put(t)`) and also taken in the same order from the “Tasks” queue (FIFO!), they are not necessarily stored in that order in the Results queue because it cannot be predicted in what order exactly the individual “worker” threads will store the results into the queue with `results.put(…)`!

---

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!

```
import multiprocessing
import time
import os


class MyProcess(multiprocessing.Process):
  def __init__(self, name, delay):
    multiprocessing.Process.__init__(self)
    self.name = name
    self.delay = delay

  def run(self):
    print('Starting process %s.' % self.name)
    print('Process ID=%i' % os.getpid())
    process_count_down(self.name, self.delay)
    print('Finished process %s.' % self.name)
```

```
def process_count_down(name, delay):
    counter = 10

    while counter:
        time.sleep(delay)
        print('Process %s counting down: %i...' % (name, counter))
        counter -= 1

### For computing the running sum, we instead define process_count_down as:
# def process_count_down(name, delay):
#     sum = 0
#
#     for x in range(100000000):
#         sum = sum + x
#
#     print('Process %s, running sum: %i...' % (name, sum))

if __name__ == '__main__':
    print('Main process: ID=%i' % os.getpid())

    p1 = MyProcess('A', 0.5)
    p2 = MyProcess('B', 0.5)
    #p3 = MyProcess('C', 0.5)
    #p4 = MyProcess('D', 0.5)

                                    # sequential:
    p1.start()                      # p1.start()
    p2.start()                      # p1.join()
    #p3.start()                     #
    #p4.start()                     # p2.start()
                                    # p2.join()
    p1.join()                       #
    p2.join()                       # #p3.start()
    #p3.join()                      # #p3.join()
    #p4.join()                      # [...]

    print('Finished.')
```

b) Runtimes:
Command line on Linux:

```
$ time python3 source-code/02-3-simple_processes-sequential.py
```

* Run 2 and 4 processes sequentially:

2 processes:	real	0m10.058s
4 processes:	real	0m20.081s

* Run 2 and 4 processes in parallel (true multiprocessing):

2 processes:	real	0m5.047s
4 processes:	real	0m5.049s

* Run 2 and 4 processes in parallel, computing the running sum of 1 to 100 million instead of waiting for 0.5 seconds:

1 process:	real	0m3.458s
2 processes:	real	0m3.485s

4 processes: real 0m3.835s

c) The **processes truly run in parallel** unless we force them to run one after the other!

d) **ALL processes compute the same running sum of 4999999950000000**! The memory is not shared, so there is no possibility of a race condition and the processes do not interfere with each other!

e) The **process IDs are all different**, so we really have different processes. If we try the same using multithreading instead of multiprocessing, we will obtain the same process ID for all the threads running within the same interpreter process (the same ID we get for the main program).

---

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.]

```
from multiprocessing import Pool
from os import getpid

def f(x):
   print("Worker process id for {0}: {1}".format(x, getpid()))
   return (x*x)

if __name__ == '__main__':
   p = Pool(5)
   #p = Pool(10)

   print(p.map(f, [1, 2, 3, 4, 5]))
   #print(p.map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
```

For Pool(5) and data points 1-5, we obtain:

```
Worker process id for 1: 25383
Worker process id for 2: 25384
Worker process id for 3: 25385
Worker process id for 4: 25386
Worker process id for 5: 25387
[1, 4, 9, 16, 25]
```

For Pool(5) and data points 1-10, we obtain:

```
Worker process id for 1: 25414
Worker process id for 2: 25415
Worker process id for 3: 25416
Worker process id for 4: 25417
Worker process id for 6: 25414
Worker process id for 5: 25418
Worker process id for 7: 25415
Worker process id for 8: 25414
Worker process id for 9: 25416
Worker process id for 10: 25417
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

=> Some **worker processes are used multiple times**; when they have performed their task, they **“return” to the pool** and can be assigned a new task!
Interesting note: in this example output the task for input 6 executed its print command before the task for input 5, although the latter was assigned first (but nearly contemporaneously).

For Pool(10) and data points 1-5, we obtain:

```
Worker process id for 1: 25444
Worker process id for 2: 25445
Worker process id for 3: 25446
Worker process id for 4: 25447
Worker process id for 5: 25448
[1, 4, 9, 16, 25]
```

=> The additional available worker processes are simply not used.

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? ... :-)

```
import multiprocessing


def withdrawBoth(balA, balB, lockA, lockB):
    for _ in range(10000):

        # withdraw 1 from balA
        lockA.acquire()
        balA.value = balA.value - 1
        lockA.release()

        # withdraw 1 from balB
        lockB.acquire()
        balB.value = balB.value - 1
        lockB.release()
```

```
def depositBoth(balA, balB, lockA, lockB):
    for _ in range(10000):

        # deposit 1 to balA
        lockA.acquire()
        balA.value = balA.value + 1
        lockA.release()

        # deposit 1 to balB
        lockB.acquire()
        balB.value = balB.value + 1
        lockB.release()


def transfer1(balA, balB, lockA, lockB):
    for _ in range(10000):
                                                                ### deadlock :-)
        lockA.acquire()                                 # lockA.acquire()
        balA.value = balA.value – 1             # lockB.acquire()
        lockA.release()                                 # balA.value = balA.value – 1
                                                                #
        lockB.acquire()                                 # balB.value = balB.value + 1
        balB.value = balB.value + 1             # lockA.release()
        lockB.release()                                 # lockB.release()


def perform_transactions():
    # initial balance (in shared memory)
    balanceA = multiprocessing.Value('i', 100)
    balanceB = multiprocessing.Value('i', 100)

    # creating a lock object and passing it to two processes
    lockA = multiprocessing.Lock()
    lockB = multiprocessing.Lock()

    p1 = multiprocessing.Process(target=withdrawBoth, args=(balanceA, balanceB, lockA, lockB))
    p2 = multiprocessing.Process(target=depositBoth, args=(balanceA, balanceB, lockA, lockB))
    p3 = multiprocessing.Process(target=transfer1, args=(balanceA, balanceB, lockA, lockB))
    p4 = multiprocessing.Process(target=transfer1, args=(balanceB, balanceA, lockB, lockA))

    p1.start()
    p2.start()
    p3.start()
    p4.start()

    p1.join()
    p2.join()
    p3.join()
    p4.join()

    print("Current balanceA = {}".format(balanceA.value))
    print("Current balanceB = {}".format(balanceB.value))

if __name__ == "__main__":
    for _ in range(10):
```

```
    perform_transactions()
```

Why can the part highlighted in red cause a deadlock?
Because we run two processes “transfer1” with inverted accounts and locks. So it is possible that the process which transfers from A to B first acquires lockA, then the process which transfers from B to A acquires lockB (which was passed as “lockA” but actually isn’t), then the first of the two processes tries to acquire lock B which is not available … and the second process waits for lock A which is also not available anymore … both release the lock they have only after obtaining the second lock, so none of them can release their current lock nor obtain the other lock … and everything is blocked ...