Translated from Spanish to English - www.onlinedoctranslator.com

Parallel Programming: Practice 3

March 11 – 22, 2024

Answers must be delivered in a single document (pdf format). This document must include the full name of all group members. If the exercise is an implementation exercise, the source code can be delivered in separate .py files within a compressed archive.

Exercise 1

In practice 2 we implemented a producer/consumer model with threads through a list and with synchronization through semaphores. To do this we use an exercise of downloading files from an HTML page.

Classqueue.QueueThe Python standard library provides this style of queue with synchronization included within. The main parameter of the constructorqueueis the maximum number of elements allowed in the queue. As in the previous exercises, this queue must be shared between all the threads, for this it is created in the main process and passed as a parameter to the child threads.

Methods for placing and removing elementsqueue.Queueareput() andget(). Additionally, this class has a methodjoin()to wait for consumers to finish. For this, it is necessary for consumers to notify by callingtask_done()every time they process an item in the queue.

- Modify the program of the last exercise of the previous practice so that it uses an objectqueue.Queueinstead of the list and traffic lights.

  Note: depending on the version of Python and the operating system, the program may not terminate, in which case the threads must be created with the parameterdaemon=True.

Exercise 2

As we have already seen in past practices, when several threads modify a common variable it is very likely that abnormal results will occur.

(race conditions). To avoid these errors, synchronization primitives are used so that only one thread updates the variable. This code that should not be executed in parallel is called a critical section. The typical Python primitive for critical sections isLockwhich has a couple of methods acquire()andrelease()like traffic lights. In this way the code has the form:

```
lock.acquire()
. . .
# sectioneithern crYoethics
. . .
lock.release()
```

Python allows for clearer syntax usingwith:

```
with lock:
    . . .
    # sectioneithern crYoethics
    . . .
```

- Modify the program from the previous exercise so that the updates of the variables total_fileandbytes_totalare synchronized with an objectLock.

  Note: Remember that a Lock is equivalent to using a semaphore initialized to 1, therefore, you can use either object.

Exercise 3

In addition to the synchronization objects seen so far: Lock, Semaphore and Queue. Python offers other synchronization mechanisms such as barriers.

A barrier is a simple synchronization primitive to be used by a fixed number of threads that need to wait for each other (the number of threads is indicated when creating the barrier, in the constructor). Each of the threads tries to pass the barrier by calling the methodwait()and will block until all threads have made their respective calls towait().At this point, all threads are released simultaneously.

The following code shows an example of using synchronization with barriers. In it, we have threads that represent the generation of hydrogen atoms and threads that represent the generation of oxygen atoms, and we want to simulate the generation of water molecules (H_2O) (i.e.

allows two threads of hydrogen and one of oxygen to be grouped)

```
import      time
import      random
from threading import Thread

def    Hydrogen():
       # time it takes to generate the atom
       time.sleep(random.randint(1, 4)) print("H")


def    Oxygen():
       # time it takes to generate the atom
       time.sleep(random.randint(1, 4)) print("O")


defgeneratedmolecule():

       print ("A new H20 molecule generated") time.sleep(0.5)



if __name__ == '__main__':

       threadlist = []

       for i in range(10):
              threadlist.append(Thread(target=Oxygen)) for j in range(10):

              threadlist.append(Thread(target=Hydrogen))

       for t in threadlist:
              t.start()
```

To “transform” these strands into water molecules, we have to make each strand wait until a complete molecule is ready to continue (two hydrogen strands and one oxygen strand) and then call the methodgeneratedmolecule()

- Modify the code so that using a barrier, water molecules are generated correctly. You may need, in addition to the

barrier, some other synchronization mechanism.

Exercise 4

Many times multithreaded programming requires “Communication between Threads”, that is, one thread notifies other threads when an event occurs or a certain state is reached. The simplest mechanism for communication between threads that Python provides is the “event”.

The Event object can have two states: enabledset()or disabledclear().Initially, the event is disabled. The threads can waitwait()until the event is enabled to continue execution.

In this exercise we are going to represent an online store that sells a single product. Customers can purchase the product if it is available and otherwise they have to wait until the product is restocked. The replenishment thread simulates the process of replenishment and end of the product, so so that all customers can buy it is repeated 2 times (replace product - end product - replenish product - end product).

```
import threading
import time
import random

available_product=False

def client(id):
    global product_available
    time.sleep(random.randint(0, staggered 10)) # Simulate
arrival of customers
    print("Customer ",id,"trying if product_available: buy...")

        print("Customer", id, "product purchased!") else:

        print("Customer ",id,"product exhausted.
Waiting for replenishment.")


def replenisher():
    global available_product
```

```
        for i in range(2):
                print("Replacing                product...")
                time.sleep(4)            # Simulate available replacement time...")
                print("Product
                available_product                =True

                time.sleep(2)            # Simulate        end of stock
                print("End product...") available_product =
                False

def main():

        # We create threads to simulate several clients
trying to buy at the same time
        client_threads = []
        for i in range(5):
                client_thread                                                    =
threading.Thread(target=client,                          args=(i,))
                client_threads.append(client_thread) client_thread.start()



        # We simulate the replacement process of the product thread_reponedor
                                                                                 =
threading.Thread(target=replacer)
        replenisher_thread.start()

        # We wait for all threads to finish for client_thread in client_threads:

                client_thread.join()

        replenisher_thread.join()

if __name__ == "__main__":
        main()
```

- When you run the code you will see that it is not working as expected. Explain why and add an event to the code to make the store work as described in the exercise.

Note: the times of thesleepfor the code to work. The problem must be solved by adding an event.