Parallel Programming: Practice 2

February 26 –March 1, 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

Although Python incorporates support for multi-threaded execution, due to the GIL it does not provide good performance for calculus-based problems. However, parallelism is obtained when threads perform input/output. In this practice we are going to parallelize a program that downloads all the files referenced by an HTML page using a producer/consumer model (one producer, several consumers). You can start from this sequential version:

```
import   you
import   urllib.request
import   urllib.parse
import   re
import   shutil
import   time
import   threading

total_file = 0
bytes_total = 0

def   download_file(address): global total_file,
      total_file bytes id =
      threading.current_thread().name total_file
                        + = 1
      print(id,        total_file, address)
      s = address.split('/') file = 'download/' +
      s[-1] try:

            i = urllib.request.urlopen(address) with open(file,
            'wb') as f:
                shutil.copyfileobj(i, f) bytes_total
                += f.tell() except Exception as
                err:
            print(err)

def   download_html(root_url, urls):
      r = urllib.request.urlopen(root_url) html =
      r.read().decode('utf-8', 'ignore')
      for m in re.finditer(r'src\w*=\w*"([-_./0-9a-zA-Z]*)"', html, re.I):

            address = urllib.parse.urljoin(root_url, m.group(1)) urls.append(address)



def   main():
      root_url = 'http://www.uv.es' if not
      os.path.exists('download'):
            os.mkdir('download')

      urls = []

      t1 = time.time()
      download_html(root_url, urls) for
      address in urls:
            download_file(address) t2 =
            time.time()
      total_time = t2 - t1

      print('---------------------') print('Time', total_time) print('Files',
      total_file) print('MBytes', total_bytes / (1024.0**2))



      print('Bandwidth (MBit/s)', (total_bytes * 8 / (1024.0 **
2))
                                                  /total_time)

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

- Run the program and check that it works by looking at the contents of the download directory. Note the execution time.

Exercise 2

To create a thread in Python, use the functionthreading.Thread.This routine has as its most important parameterstargetwhich is the function that the thread will execute andargs which indicates the parameters to this function. threading.Threadreturns an object that identifies the thread and that will be needed for other operations:start()that starts the thread andjoin()waiting for the thread to end.

- Make a parallel version of the program in exercise 1 where each call to download_file is executed in a different thread. Use the function:

```
threading.Thread(target=download_file, args=(address,))
```

  Don't forget to start the threads with start() and add the threads to a list. At the end of the main program you have to wait for the threads to finish (otherwise the main program will end early) with a call to join().

- Compare the times of the sequential version with the parallel one.

Exercise 3

A problem with the previous exercise is that it creates as many threads as there are downloaded files. This is usually not very efficient since the cost of creating threads is not negligible. Additionally, web servers often limit the maximum number of connections, penalizing clients who try to use too many. Additionally, if there is too little time between connections the server stops responding. This is done to prevent denial of service (DoS) attacks.

Alternatively, the main program can create a fixed number of consumer threads (for example 4) and a producer. The producer puts the addresses into a list and is the same as the previous exercise, but now runs in its own thread:

```
p = threading.Thread(target=download_html, args=(root_url, urls))
```

Consumers take addresses from the list and download them one by one. Consumers process and remove elements from the address list so that downloads are distributed among them. This is what is known in parallel programming as the producer-consumer model. Consumers work as long as the list is not empty.

- Modify the program from the previous exercise so that it uses a fixed number of consumer threads (for example 4) and a producer.
- Compare the execution time with previous versions.

Exercise 4

The program in the previous exercise requires that the consumers do not start until the producer finishes, since if they start before, since the list has no elements, they would end the function without doing anything. To achieve greater parallelism, it is desirable that as soon as the list has an element to download, consumers can start working. One solution is to use synchronization primitives so that threads are deactivated when there is no data in the list and reactivated when the list has elements. A primitive that can be used in this case is a semaphore initialized to zero (wait semaphore):

semaphore = threading.Semaphore(0)

The producer unlocks the traffic light withsemaphore.release()every time you place an item in the list. Consumers wait with semaphore.acquire()before taking an item from the list, download it and repeat the process until the list is empty.

- Modify the program from the previous exercise so that it uses a semaphore to block the consuming threads. Remember that consumer threads must start before the producer ends.

  Make sure that the main program terminates, meaning that all consuming threads terminate and are not stuck at the semaphore when the list is emptied. To do this, once the producer has filled out the list, the main program must make asemaphore.release() extra so that each consuming thread can finish.

Exercise 5

In the previous version of the program, the address list can grow arbitrarily, which requires more memory. However, in some applications it is necessary to limit the length of the list to avoid excessive memory consumption. To do this, a second semaphore initialized to the maximum size of the queue is added.

semaphoreMaxQueue = threading.Semaphore(max)

The producer decrements the semaphore each time it adds an element, and consumers increment it each time they process an element in the list.

- Add a semaphore to the previous program so that the queue never has more than 10 elements.