Programación Paralela: Práctica 2

26 febrero –1 marzo 2024

Las respuestas deben entregarse en un único documento (formato pdf). Este documento debe incluir el nombre completo de todos los integrantes del grupo. Si el ejercicio es de implementación, el código fuente se puede entregar en ficheros .py independientes dentro de un archivo comprimido.

Ejercicio 1

Aunque Python incorpora soporte para ejecución con múltiples hilos, debido al GIL no obtiene buenas prestaciones para problemas basados en cálculo. Sin embargo, sí quese obtiene paralelismo cuando los hilos realizan entrada/salida. En esta práctica vamos a paralelizar un programa que descarga todos los ficheros referenciados por una página HTML utilizando un modelo productor/consumidor (un productor, varios consumidores).Puedes partir de esta versión secuencial:

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

fich_total = 0
bytes_total = 0

def bajar_fichero(direccion):
    global fich_total, bytes_total
    id = threading.current_thread().name
    fich_total += 1
    print(id, fich_total, direccion)
    s = direccion.split('/')
    fichero = 'download/' + s[-1]
    try:
        i = urllib.request.urlopen(direccion)
        with open(fichero, 'wb') as f:
```

```
            shutil.copyfileobj(i, f)
            bytes_total += f.tell()
    except Exception as err:
        print(err)

def bajar_html(url_raiz, urls):
    r = urllib.request.urlopen(url_raiz)
    html = r.read().decode('utf-8', 'ignore')
    for m in re.finditer(r'src\w*=\w*"([-_./0-9a-zA-Z]*)"', html,
re.I):
        direccion = urllib.parse.urljoin(url_raiz, m.group(1))
        urls.append(direccion)

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

    urls = []

    t1 = time.time()
    bajar_html(url_raiz, urls)
    for direccion in urls:
        bajar_fichero(direccion)
    t2 = time.time()
    tiempo_total = t2 - t1

    print('----------------------')
    print('Tiempo', tiempo_total)
    print('Ficheros', fich_total)
    print('MBytes', bytes_total / (1024.0 ** 2))
    print('Ancho de banda (MBit/s)', (bytes_total * 8 / (1024.0 **
2))
                                      / tiempo_total)

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

- Ejecuta el programa y comprueba que funciona mirando el contenido del directorio download. Anota el tiempo de ejecución.

Ejercicio 2

Para crear un hilo en Python se utiliza la función `threading.Thread`. Esta rutina tiene como parámetros más importantes `target` que es la función que ejecutará el hilo y `args` que indica los parámetros a esta función. `threading.Thread` devuelve un objeto queidentifica al hilo y que hará falta para otras operaciones: `start()` que pone en marchael hilo y `join()` que espera a que el hilo acabe.

- Haz una versión paralela del programa del ejercicio 1 donde cada llamada a bajar_fichero se ejecute en un hilo distinto. Utiliza la función:

  ```
  threading.Thread(target=bajar_fichero, args=(direccion ,))
  ```

  No olvides arrancar los hilos con start() y añade los hilos a una lista. Al final del programa principal hay que esperar a que los hilos acaben (si no el programa principal terminará antes de tiempo) con una llamada a `join()`.
- Compara los tiempos de la versión secuencial con la paralela.

Ejercicio 3

Un problema del ejercicio anterior es que crea tantos hilos como ficheros descargados. Esto no suele ser muy eficiente ya que el coste de crear hilos no es despreciable. Además, los servidores web suelen limitar el número máximo de conexiones, penalizandoa los clientes que intentan utilizar demasiadas. Además, si hay muy poco tiempo entre conexiones el servidor deja de responder. Esto se hace para evitar ataques de denegaciónde servicio (DoS).

Como alterativa, el programa principal puede crear un número fijo de hilos consumidores (por ejemplo 4) y un productor. El productor coloca las direcciones en una lista y es el mismo que el ejercicio anterior, pero ahora se ejecuta en su propio hilo:

```
p = threading.Thread(target=bajar_html, args=(url_raiz,
urls))
```

Los consumidores van cogiendo direcciones de la lista y las bajan de una en una. Los consumidores van procesando y quitando elementos de la lista de direccionesde manera que las descargas se repartan entre ellos. Esto es lo que se conoce en programación paralela como el modelo productor-consumidor. Los consumidores trabajan mientras la lista no este vacía.

- Modifica el programa del ejercicio anterior para que utilice un número fijo de hilos consumidores (por ejemplo 4) y un productor.
- Compara el tiempo de ejecución con las versiones anteriores.

Ejercicio 4

El programa del ejercicio anterior requiere que los consumidores no empiecen hasta que el productor termina, ya que, si empiezan antes, como la lista no tiene elementos, terminarían la función sin hacer nada. Para conseguir mayor paralelismo, es deseable que en el momento en el que la lista tenga algún elemento para descargar los consumidores puedan empezar a trabajar. Una solución es utilizar primitivas de sincronización para que los hilos se desactiven cuando no hay datos en la lista y se reactiven cuando la lista tenga elementos. Una primitiva que se puede utilizar en este caso es un semáforo inicializado a cero (semáforo de espera):

```
semaforo = threading.Semaphore(0)
```

El productor desbloquea el semáforo con `semaforo.release()` cada vez que coloca un elemento en la lista. Los consumidores esperan con `semaforo.acquire()` antes de coger un elemento de la lista, descargarlo y repetir el proceso hasta que la lista este vacía.

- Modifica el programa del ejercicio anterior para que utilice un semáforo para bloquear los hilos consumidores. Recuerda que los hilos consumidores deben empezar antes de que termine el productor.

  Asegúrate de que el programa principal termina, es decir que todos los hilos consumidores terminan y no se quedan bloqueados en el semáforo cuando la lista se vacía. Para ello, una vez que el productor ha llenado la lista, el programa principal debe hacer un `semaforo.release()` extra para que cada hilo consumidor pueda terminar.

Ejercicio 5

En la versión anterior del programa, la lista de direcciones puede crecer arbitrariamente, lo que supone mayor gasto de memoria. Sin embargo, en algunas aplicaciones es necesario limitar la longitud de la lista para evitar un consumo de memoria excesivo. Para ello se añade un segundo semáforo inicializado al tamaño máximo de la cola.

```
semaforoMaxCola = threading.Semaphore(max)
```

El productor decrementa el semáforo cada vez que añade un elemento y los consumidoreslo incrementan cada vez que procesan un elemento de la lista.

- Añade al programa anterior un semáforo de manera que la cola nunca tenga más de 10 elementos.