Programación Paralela: Práctica 3

11 – 22 de 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

En la práctica 2 implementamos un modelo productor/consumidor con hilos mediante una lista y con sincronización mediante semáforos. Para ello utilizamos un ejercicio de descarga de los ficheros de una página HTML.

La clase `queue.Queue` de la biblioteca estándar de Python proporciona este estilo de cola con la sincronización incluida dentro. El parámetro principal del constructor `Queue` es el número de elementos máximos permitidos en la cola. Al igual que en los ejercicios anteriores esta cola debe estar compartida entre todos los hilos, para ello se crea en el proceso principal y se pasa como parámetro a los hilos hijos.

Los métodos para colocar y quitar elementos de `queue.Queue` son `put()` y `get()`. Además, esta clase tiene un método `join()` para esperar a que los consumidores terminen. Para ello es necesario que los consumidores avisen mediante una llamada a `task_done()` cada vez que procesan un elemento de la cola.

- Modifica el programa del último ejercicio de la práctica anterior para que utilice un objeto `queue.Queue` en lugar de la lista y los semáforos.

  Nota: dependiendo de la versión de Python y del sistema operativo es posible que el programa no termine, en ese caso hay que crear los hilos con el parámetro `daemon=True`.

Ejercicio 2

Como ya hemos visto en prácticas pasadas, cuando varios hilos modifican una variable común es muy probable que se produzcan resultados anómalos

(condiciones de carrera). Para evitar estos errores se utilizan primitivas de sincronización para que solamente un hilo actualice la variable. Este código que no se debe ejecutar en paralelo se denomina sección crítica. La primitiva típica de Python para secciones críticas es `Lock` que tiene un par de métodos `acquire()` y `release()` como los semáforos. De esta manera el código tiene la forma:

```
lock.acquire()
...
#sección crítica
...
lock.release()
```

Python permite una sintaxis más clara usando `with`:

```
with lock:
      ...
      #sección crítica
      ...
```

- Modifica el programa del ejercicio anterior de manera que las actualizaciones de las variables `fich_total` y `bytes_total` estén sincronizadas con un objeto `Lock`.

  Nota: Recuerda que un Lock es equivalente a utilizar un semáforo inicializado a 1, por tanto, puedes utilizar cualquiera de los dos objetos.

Ejercicio 3

Además de los objetos de sincronización vistos hasta ahora: Lock, Semáforo y Cola. Python ofrece otros mecanismos de sincronización como son las barreras.

Una barrera es una primitiva de sincronización simple para ser usada por un número fijo de hilos que necesitan esperarse entre ellos (el número de hilos se indica al crear la barrera, en el constructor). Cada uno de los hilos intenta pasar la barrera llamando al método `wait()` y se bloqueará hasta que todos los hilos hayan hecho sus respectivas llamadas a `wait()`. En este punto, todos los hilos son liberados simultáneamente.

El siguiente código muestra un ejemplo de uso de sincronización con barreras. En él, tenemos hilos que representan la generación de átomos de hidrógeno e hilos que representan la generación de átomos de oxígeno, y queremos simular la generación de moléculas de agua (H_2O) (es decir que

permita agrupar dos hilos de hidrógeno y uno de oxígeno)

```
import time
import random
from threading import Thread

def Hidrogeno():
    #tiempo que tarda en generarse el atomo
    time.sleep(random.randint(1, 4))
    print("H")

def Oxigeno():
    #tiempo que tarda en generarse el atomo
    time.sleep(random.randint(1, 4))
    print("O")

def moleculagenerada():

    print ("Generada una nueva molécula de H20")
    time.sleep(0.5)


if __name__ == '__main__':

    listahilos = []

    for i in range(10):
        listahilos.append(Thread(target=Oxigeno))
    for j in range(10):
        listahilos.append(Thread(target=Hidrogeno))

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

Para "transformar" estos hilos en moléculas de agua, tenemos que hacer que cada hilo espere hasta que una molécula completa esté lista para continuar (dos hilos de hidrógeno y un hilo de oxígeno) y entonces se llame al método `moleculagenerada()`

- Modifica el código para que utilizando una barrera se generen de forma correcta las moléculas de agua. Es posible que necesites, además de la

barrera, algún otro mecanismo de sincronización.

Ejercicio 4

Muchas veces la programación multihilo requiere "Comunicación entre Hilos", es decir, que un hilo notifique a otros hilos cuando se produce algún evento o se alcanza cierto estado. El mecanismo más sencillo de comunicación entre hilos que Python proporciona es el "evento".

El objeto Event puede tener dos estados: habilitado `set()` o deshabilitado `clear()`. Inicialmente, el evento está deshabilitado. Los hilos pueden esperar `wait()` hasta que el evento esté habilitado para continuar su ejecución.

En este ejercicio vamos a representar una tienda en línea que vende un único producto. Los clientes pueden comprar el producto si está disponible y, de lo contrario, tienen que esperar hasta que se reponga el producto. El hilo reponedor simula el proceso de reposición y fin del producto, por eso para que todos los clientes puedan comprar se repite 2 veces (repone producto- fin producto – repone producto – fin producto).

```
import threading
import time
import random

producto_disponible=False

def cliente(id):
    global producto_disponible
    time.sleep(random.randint(0,   10))      #   Simular
llegada escalonada de clientes
    print("Cliente ",id,"intentando comprar...")
    if producto_disponible:
        print("Cliente ",id, "producto comprado!")
    else:
        print("Cliente     ",id,"producto     agotado.
Esperando reposición.")


def reponedor():
    global producto_disponible
```

```
    for i in range(2):
        print("Reponiendo producto...")
        time.sleep(4)  # Simular tiempo de reposición
        print("Producto disponible...")
        producto_disponible = True

        time.sleep(2)  # Simular fin existencias
        print("Fin producto...")
        producto_disponible = False

def main():

    # Creamos hilos para simular varios clientes
intentando comprar al mismo tiempo
    hilos_clientes = []
    for i in range(5):
        hilo_cliente                                    =
threading.Thread(target=cliente, args=(i,))
        hilos_clientes.append(hilo_cliente)
        hilo_cliente.start()

    # Simulamos el proceso de reposición del producto
    hilo_reponedor                                      =
threading.Thread(target=reponedor)
    hilo_reponedor.start()

    # Esperamos a que todos los hilos terminen
    for hilo_cliente in hilos_clientes:
        hilo_cliente.join()

    hilo_reponedor.join()

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

- Al ejecutar el código verás que el funcionamiento no es el esperado. Explica por qué y añade un evento al código para conseguir que la tienda funcione como se ha descrito en el ejercicio.

Nota: no se deben modificar los tiempos de los `sleep` para que el código funcione. Hay que solucionar el problema añadiendo un evento.