Ejercicio 1:

Se tiene el siguiente código en Python que simula la actualización de una variable compartida llamada counter a través de varios hilos. Cada hilo ejecuta una función que incrementa el valor de counter en un bucle. La cantidad total esperada al final de la ejecución es la suma de todos los incrementos realizados por cada hilo.

```python
import threading
counter = 0
def increment():
    global counter
    for _ in range(1000):
        counter += 1

threads = [threading.Thread(target=increment) for _ in range(10)]

for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

print("Counter:", counter)
```

a) ¿Cuál sería el valor esperado de la variable counter al finalizar la ejecución? Tras ejecutar el código varias veces, ¿observa siempre el mismo resultado? Justifique su respuesta.

- El **valor esperado** de la variable counter sería **10000** esto es porque cada uno de los 10 hilos ejecuta la función increment, la cual incrementa counter en 1 un total de 1,000 veces. Así que el incremento debería ser 10×1,000=10000.
- **Sí, siempre obtengo el valor 10000** al ejecutar el programa varias veces, Porque es un problema muy básico que al ejecutarlo en nuestros ordenadores no observamos la diferencia en consola y da como he dicho antes siempre el mismo valor si esto lo probamos en un ordenador más antiguo veríamos cambios en los resultados, también el GIL de Python reduce la probabilidad de estos errores debidos a condiciones de carrera al limitar la ejecución a un hilo cada vez.

```
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1>
```

b) Explique, paso a paso, por qué el código podría no producir siempre el mismo resultado al ejecutarse. Detalle los posibles comportamientos que pueden ocurrir durante la ejecución del programa.

El código podría no producir siempre el mismo resultado debido a problemas de concurrencia al ejecutar varios hilos sobre la misma variable compartida, counter, sin una sincronización adecuada. **La explicación paso a paso es:**

1. **Inicio de los hilos**: Se crean 10 hilos, y cada uno ejecuta la función increment, que intenta sumar 1 a counter 1,000 veces. Como counter es una variable compartida entre todos los hilos, su valor cambia con cada incremento.
2. **Acceso simultáneo**: Al no tener sincronización, los hilos pueden acceder y modificar counter al mismo tiempo. Esto significa que dos o más hilos podrían leer el mismo valor de counter antes de que otro hilo lo actualice.
3. **Condición de carrera**: Una condición de carrera ocurre si dos o más hilos leen el mismo valor de counter y luego intentan incrementarlo de forma independiente. Por ejemplo, el Hilo A y el Hilo B podrían leer counter cuando su valor es 1000, incrementar cada uno en 1 (pensando que el valor ahora es 1001), y sobrescribir el valor del otro, resultando en que counter solo aumente en 1 en lugar de en 2.
4. **Pérdida de incrementos**: Como resultado de estas condiciones de carrera, algunos incrementos pueden perderse porque un hilo sobrescribe el valor actualizado por otro. Esto hace que el valor final de counter al final de la ejecución pueda ser menor a 10000.

Los posibles comportamientos que pueden ocurrir durante la ejecución son:

1. **Resultado esperado (10000)**: Lo que nos ocurre a nosotros que pasa porque los hilos se ejecutan en un orden ideal, sin interferencias.
2. **Pérdida de incrementos**: Dos o más hilos pueden leer el mismo valor de counter, incrementarlo y sobrescribir sus cambios, haciendo que algunos incrementos se pierdan y el valor final sea menor a 10,000.

c) En este código, ¿existe algún tipo de interacción entre hilos que podría llevar a resultados inesperados? Describa, de forma detallada, cuál podría ser el origen de este comportamiento.

Sí, en este código existe una interacción entre hilos que puede llevar a resultados inesperados debido a una condición de carrera. Esta ocurre porque todos los hilos acceden y modifican simultáneamente la variable compartida counter sin sincronización. Como resultado, dos o más hilos pueden leer el

mismo valor de counter, incrementarlo, y sobrescribir sus cambios, haciendo que algunos incrementos se pierdan y que el valor final de counter sea menor al esperado.

d) En el contexto de este código, ¿qué se entiende por "operación atómica"? ¿Por qué podría ser importante este concepto para garantizar la correcta actualización de counter?

Una operación atómica es una unidad de trabajo que es indivisible y se garantiza que se ejecutará como una operación única, coherente e ininterrumpida. En esencia, una operación atómica es un conjunto de instrucciones que se ejecutan secuencialmente sin ninguna interrupción o interferencia de otros procesos o subprocesos concurrentes.

Este concepto es importante para garantizar la correcta utilización del counter porque cuando varios procesos o subprocesos intentan modificar un recurso compartido simultáneamente (variable counter), pueden producirse condiciones de carrera, lo que genera resultados impredecibles y erróneos. Las operaciones atómicas lo que hacen es proporcionar un mecanismo para mitigar estos problemas al garantizar que se acceda al recurso compartido de una manera mutuamente excluyente, evitando conflictos y asegurando que las operaciones se ejecuten de forma atómica.

https://startup-house.com/glossary/atomic-operation

e) Observe la función increment. ¿Identifica algún momento específico en el que el acceso a counter podría producir resultados inesperados cuando varios hilos están ejecutando esta función? Justifique su respuesta.

Si, el momento específico en que el acceso a counter podría producir resultados inesperados ocurre durante la lectura y escritura de counter en la línea en la que el counter se incrementa, ya que aquí se lee, se incrementa y se escribe el nuevo valor en el counter.

Cuando varios hilos ejecutan esta operación al mismo tiempo, pueden leer el mismo valor inicial de counter, realizar sus incrementos de forma independiente y luego escribir su resultado sin tener en cuenta los cambios hechos por otros hilos. Esto resulta en sobre escritura, donde el incremento de un hilo reemplaza al de otro, y se pierden incrementos en counter provocando que el valor final de counter sea menor al esperado.

f) ¿Considera que el código actual requiere algún ajuste para garantizar un resultado consistente al finalizar la ejecución? ¿Existen técnicas de sincronización que podrían aplicarse en este contexto? Si es así, describa alguna de ellas y proporciona una posible implementación en código que permita alcanzar un resultado consistente al finalizar.

El código actual siempre sale lo mismo por lo que no creo que haga falta ningún ajuste, aunque al no tener sincronización el resultado podría salir diferente por como Python maneja los hilos, algunas técnicas de sincronización que se podrían aplicar a nuestro código para asegurarse de que eso no ocurra son:

1. **Lock ->** lo que hace es que solo un hijo acceda a counter a la vez, garantizando que los incrementos sean seguros, utilizaríamos threading.Lock para así proteger la sección crítica.

```python
import threading

counter = 0
lock = threading.Lock()

def increment():
    global counter
    for _ in range(1000):
        with lock:
            counter += 1

threads = [threading.Thread(target=increment) for _ in range(10)]

for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

print("Counter:", counter)
```

```
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
PS C:\Users\marin\OneDrive\Desktop\PEC1_SD_Marina_Amaya\Ejercicio1> python ejercicio1_lock.py
Counter: 10000
```