UNIVERSIDAD NACIONAL DE MOQUEGUA

Facultad de Ingeniería y Arquitectura

Escuela Profesional de Ingeniería de Sistemas e Informática

Ejercicios sobre Programación en Memoria Compartida y Hilos

Informe laboratorio N° 1

Estudiante:
Gersael Mathias Rojas Quijano

Profesores:
Honorio Apaza Alanoca

15 de mayo de 2025

Índice

1. Introducción

1.1. Motivación y Contexto

En la era actual de la computación de alto rendimiento, la capacidad de aprovechar múltiples núcleos de procesamiento se ha vuelto esencial para desarrollar aplicaciones eficientes. La programación en memoria compartida, mediante el uso de hilos, permite a los desarrolladores dividir tareas complejas en unidades más pequeñas que se pueden ejecutar en paralelo, mejorando así la velocidad y la eficiencia del procesamiento.

1.2. Objetivo general

Desarrollar habilidades prácticas en programación concurrente utilizando memoria compartida y manejo de hilos en Python, a través de la resolución de ejercicios que abordan problemas reales de sincronización, paralelismo y acceso a recursos compartidos.

1.3. Objetivos específicos

- Comprender el funcionamiento del modelo de memoria compartida en la programación concurrente.
- Implementar y gestionar hilos en Python para la ejecución simultánea de tareas.
- Analizar el comportamiento de los programas en paralelo y medir el impacto del uso de hilos en el rendimiento de las aplicaciones.

1.4. Justificación

La programación en memoria compartida con hilos es clave en el desarrollo de software moderno, ya que permite aprovechar múltiples núcleos de procesamiento. Este trabajo ayuda a comprender los fundamentos del paralelismo, abordando tanto sus ventajas como los retos, como la sincronización y el acceso seguro a recursos. Además, promueve el pensamiento crítico y la resolución de problemas en entornos concurrentes.

2. Marco teórico

2.1. Antecedentes

El desarrollo de algoritmos paralelos se remonta a las primeras investigaciones sobre arquitecturas de procesamiento paralelo, surgidas como solución a las limitaciones de velocidad en los sistemas secuenciales. A medida que los procesadores multinúcleo se han vuelto estándar, se ha incrementado la necesidad de programar aplicaciones capaces de ejecutar múltiples tareas simultáneamente.
Python, a pesar de no ser considerado tradicionalmente como un lenguaje de alto rendimiento para paralelismo debido a su Global Interpreter Lock (GIL), ha incorporado herramientas como los módulos "threading", "multiprocessingz çoncurrent.futures", que permiten a los programadores trabajar con modelos de concurrencia y paralelismo de manera sencilla y educativa.

2.2. Marco conceptual

2.2.1. Tema 1: Algoritmos paralelos

Los algoritmos paralelos son aquellos diseñados para ejecutarse en múltiples unidades de procesamiento al mismo tiempo. Se diferencian de los algoritmos secuenciales en que dividen el problema en subproblemas que se pueden resolver simultáneamente. La eficiencia de un algoritmo paralelo se mide en términos de aceleración y escalabilidad.

2.2.2. Tema 2: Hilos en Python

Un hilo (thread) es la unidad básica de ejecución dentro de un proceso. En Python, el módulo threading permite la creación y gestión de hilos dentro de un programa, permitiendo que varias funciones se ejecuten .[en] paralelo"dentro del mismo espacio de memoria. Aunque el GIL limita la ejecución verdaderamente paralela en procesos CPU-bound, los hilos son muy útiles para tareas concurrentes, especialmente.

2.2.3. Tema 3: Sincronización de hilos

Cuando varios hilos acceden a recursos compartidos, pueden surgir problemas como condiciones de carrera, bloqueos (deadlocks) y resultados inconsistentes. Para evitar estos problemas, se utilizan mecanismos de sincronización como:

- **Locks (cerrojos):** impiden el acceso simultáneo a una sección crítica del código.
- **Semáforos:** permiten gestionar el acceso concurrente limitado a un recurso.
- **Barreras:** sincronizan la ejecución de varios hilos para que todos lleguen a un punto antes de continuar.

3. Metodología

Este trabajo adopta un enfoque cuantitativo y experimental, basado en la implementación práctica de algoritmos concurrentes utilizando hilos. La metodología se centra en la ejecución de ejercicios diseñados para reforzar el aprendizaje mediante la experiencia directa, fomentando la aplicación de conceptos teóricos en situaciones concretas. A través de pruebas controladas y análisis del comportamiento del código, se busca comprender el impacto de la concurrencia en la eficiencia y seguridad del acceso a recursos compartidos.

3.1. Enfoque Metodológico

3.2. Proceso de Resolución

El desarrollo del trabajo se realiza en varias etapas estructuradas:

1. **Revisión teórica:** Lectura y comprensión de los fundamentos sobre memoria compartida, hilos y sincronización en Python.
2. **Análisis de ejercicios:** Estudio detallado de los 10 ejercicios propuestos, identificando el objetivo de cada uno y los conceptos aplicados.
3. **Codificación:** Implementación de cada ejercicio en Python utilizando el módulo threading y otros complementos necesarios.
4. **Desarollo y Resultado:** Ejecución de los programas para verificar su correcto funcionamiento, asegurando la integridad de los datos y la ausencia de condiciones de carrera.
5. **Documentación:** Registro de resultados, análisis del comportamiento de los hilos, reflexiones sobre la eficiencia y posibles mejoras.

3.3. Instrumentos y Herramientas

- **Lenguajes de programación:** (versión 3.10 o superior)
- **Librerías:**
  - threading para la gestión de hilos.
  - time para medición de tiempos y retardos.
  - queue, semaphore, lock para sincronización y coordinación entre hilos.
- **Entornos de desarrollo:** VSCode, NetBeans o Jupyter Notebook.
- **Sistema operativo:** Windows

3.4. Resultados Esperados

Al finalizar el desarrollo de los ejercicios, se espera obtener:

- Implementación correcta de los 10 ejercicios propuestos, aplicando técnicas de programación concurrente en memoria compartida.
- Evidencia de mejoras en el rendimiento o en la claridad del flujo de ejecución de tareas mediante el uso adecuado de hilos.
- Desarrollo de competencias clave en el uso de estructuras de sincronización como locks, semáforos y barreras, aplicadas a casos reales.

4. Propuesta

4.1. Ejercicio Básico de Hilos: Suma de Números

4.1.1. Análisis del ejercicio

Este ejercicio busca aplicar la programación con hilos dividiendo la suma de los primeros 1,000,000 de números naturales entre dos hilos. Cada hilo calcula una mitad y luego se combinan los resultados usando sincronización para evitar condiciones de carrera. Permite reforzar el uso de threading y mecanismos como Lock para asegurar la correcta manipulación de variables compartidas.

4.1.2. Codificacion:

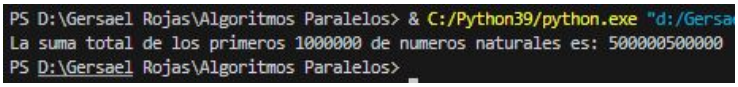
```
import threading
suma_total = 0
lock = threading.Lock()
def sumar_rango(inicio, fin):
    global suma_total
    suma_parcial = sum(range(inicio, fin))
    with lock:
        suma_total += suma_parcial

n=1_000_000
mitad=n//2
hilo1=threading.Thread(target=sumar_rango, args=(1, mitad+1))
hilo2=threading.Thread(target=sumar_rango, args=(mitad + 1,n+1))

hilo1.start()
hilo2.start()

hilo1.join()
hilo2.join()

print("La suma total de los primeros 1000000 de numeros naturales es:",
   suma_total)
```

4.1.3. Desarrollo y Resultados:

- **Desarrollo:** Para resolver el ejercicio, se utilizó la librería `threading` de Python, creando dos hilos que procesan mitades iguales de un rango de 1000000 de números naturales. Se empleó un Lock para sincronizar el acceso a la variable compartida "suma-total", evitando condiciones de carrera. Cada hilo ejecuta una función que suma su segmento y agrega el resultado parcial a la variable global de forma segura. Una vez finalizados ambos hilos, el programa imprime la suma total. El resultado fue

verificado con la fórmula matemática de la suma de números naturales, y coincidió exactamente.

- **Resultado:**



El programa devuelve correctamente la suma de los primeros 1000000 números naturales: 500000500000. Esto demuestra que la sincronización con Lock funcionó correctamente, ya que no hubo pérdida de datos ni condiciones de carrera.

4.2. Ejercicio de Sincronización: Contador de Primos

4.2.1. Análisis del ejercicio

Este ejercicio busca desarrollar un programa concurrente en Python que utilice dos hilos para trabajar sobre una lista compartida de números. El primer hilo cuenta cuántos números son primos, mientras que el segundo calcula la suma de esos mismos primos. Se emplea un Lock para sincronizar el acceso a la lista y evitar condiciones de carrera durante las operaciones concurrentes.

4.2.2. Codificacion:

```python
import threading

numeros=list(range(2, 10001))
primos=[]
lock=threading.Lock()
def es_primo(n):
    if n<2:
        return False
    for i in range(2,int(n**0.5)+1):
        if n%i==0:
            return False
    return True
def contar_primos():
    for numero in numeros:
        if es_primo(numero):
            with lock:
                primos.append(numero)

def sumar_primos():
    while True:
        with lock:
            if len(primos)==len([n for n in numeros if es_primo(n)]):
```

```
                suma=sum(primos)
                print(f"Suma de n meros primos: {suma}")
                break

hilo1 = threading.Thread(target=contar_primos)
hilo2 = threading.Thread(target=sumar_primos)
hilo1.start()
hilo2.start()

hilo1.join()
hilo2.join()

print(f"Numeros primos encontrados: {len(primos)}")
```

4.2.3. Desarrollo y Resultados:

- **Desarollo:** Este ejercicio implementa dos hilos en paralelo que operan sobre una lista compartida de números del 2 al 10000. Uno de los hilos identifica y almacena los números primos en una lista protegida con Lock, mientras que el segundo hilo espera que se haya completado el análisis para luego calcular y mostrar la suma total de los números primos encontrados.

  Se utilizó la estructura "threading.Lock()"para evitar condiciones de carrera al modificar la lista compartida primos. La función .es-primo().optimiza el proceso de verificación de números primos utilizando divisiones hasta la raíz cuadrada del número.

- **Resultado:**

```
PS D:\Gersael Rojas\Algoritmos Paralelos> & C:
Suma de números primos: 5736396
Números primos encontrados: 1229
PS D:\Gersael Rojas\Algoritmos Paralelos>
```

  El programa imprime correctamente la cantidad total de números primos encontrados en el rango y su suma. Esto valida el correcto funcionamiento de la sincronización entre hilos y la manipulación segura de recursos compartido

4.3. Ejercicio de Condiciones de Carrera: Suma en Paralelo

4.3.1. Análisis del ejercicio

Este ejercicio consiste en sumar los elementos de una lista de 10 millones de números utilizando dos hilos en Python, sin aplicar sincronización. Cada hilo procesa la mitad de la lista y almacena su resultado en una variable compartida, lo que puede generar errores debido a condiciones de carrera. La ausencia de mecanismos como Lock puede provocar

resultados incorrectos o inconsistentes, evidenciando la importancia de controlar el acceso concurrente a variables compartidas.

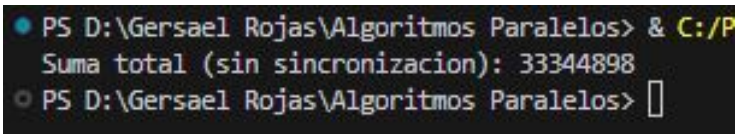
4.3.2. Codificacion:

```python
import threading
import random

datos = [random.randint(1,10) for _ in range(10_000_000)]
suma_total = 0

def sumar_parte(inicio, fin):
    global suma_total
    for i in range(inicio, fin):
        suma_total+=datos[i]
mitad = len(datos) // 2
hilo1 = threading.Thread(target=sumar_parte, args=(0, mitad))
hilo2 = threading.Thread(target=sumar_parte, args=(mitad, len(datos)))

hilo1.start()
hilo2.start()

hilo1.join()
hilo2.join()

print(f"Suma total (sin sincronizacion): {suma_total}")
```

4.3.3. Desarrollo y Resultados:

- **Desarollo:** Este ejercicio demuestra el problema clásico de las condiciones de carrera. Se genera una lista de 10 millones de números aleatorios entre 1 y 10. Luego, dos hilos suman cada mitad de la lista sin ningún mecanismo de sincronización (Lock), accediendo y modificando la variable global "suma-total"de forma concurrente.

- **Resultados:**



  La falta de sincronización sobre "suma-total"provoca que cuando un hilo está leyendo y escribiendo al mismo tiempo que el otro, los resultados de la suma se corrompen. Esto ejemplifica un error de condición de carrera, un problema crítico en la programación concurrente.

4.4. Ejercicio de Sincronización con Hilos: Cálculo de Media

4.4.1. Análisis del ejercicio

Este ejercicio plantea el cálculo de la media de una lista de 1 millón de números utilizando dos hilos en Python. Cada hilo suma una mitad de la lista y luego se combinan ambos resultados, asegurando la correcta manipulación de las variables compartidas mediante el uso de Lock. Esto permite practicar el reparto de carga de trabajo y la sincronización en memoria compartida para obtener un resultado preciso y libre de condiciones de carrera.

4.4.2. Codificacion:

```python
import threading
import random

datos=[random.randint(1, 100) for _ in range(1_000_000)]
suma_total=0
lock=threading.Lock()
def calcular_suma(inicio, fin):
    global suma_total
    suma_local = sum(datos[inicio:fin])
    with lock:
        suma_total += suma_local

mitad = len(datos) // 2
hilo1 = threading.Thread(target=calcular_suma, args=(0, mitad))
hilo2 = threading.Thread(target=calcular_suma, args=(mitad, len(datos)))

hilo1.start()
hilo2.start()

hilo1.join()
hilo2.join()

media=suma_total/len(datos)
print(f"Media calculada: {media:.2f}")
```

4.4.3. Desarrollo y Resultados:

- **Desarollo:** En este programa se utiliza programación concurrente con hilos para calcular la media de una lista de un millón de números. Se divide la lista en dos partes, y cada hilo se encarga de calcular la suma de una mitad. Luego, las sumas parciales se combinan utilizando un Lock para evitar condiciones de carrera al modificar la variable compartida "suma-total".
- **Resultados:**

```
PS D:\Gersael Rojas\Algoritmos Paralelos> & C
Media calculada: 50.53
PS D:\Gersael Rojas\Algoritmos Paralelos>
```

Este ejercicio refuerza el uso de sincronización en operaciones compartidas y muestra cómo dividir tareas para mejorar el rendimiento con hilos sin comprometer la precisión.

4.5. Ejercicio Avanzado: Ordenamiento de una Lista con Hilos

4.5.1. Análisis del ejercicio

Este ejercicio propone implementar un programa en Python que ordene una lista de números utilizando múltiples hilos. Cada hilo se encarga de ordenar una sublista, y posteriormente los resultados se combinan mediante un algoritmo de fusión como merge sort. El ejercicio permite explorar la paralelización del proceso de ordenamiento y la coordinación entre hilos para lograr una lista final correctamente ordenada.

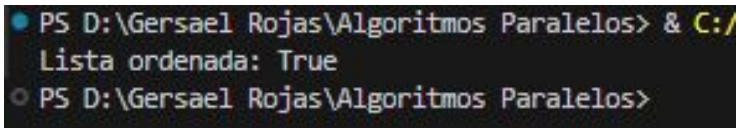
4.5.2. Codificacion:

```python
import threading
import random

lista_original=[random.randint(1,1000) for _ in range(1000000)]
sublistas=[]
resultados=[]

num_hilos=4
tama o_parte=len(lista_original)//num_hilos

def ordenar_parte(sublista, indice):
    resultado=sorted(sublista)
    resultados[indice]=resultado

for i in range(num_hilos):
    inicio=i*tama o_parte
    fin=(i + 1)*tama o_parte if i<num_hilos-1 else len(lista_original)
    sublistas.append(lista_original[inicio:fin])
    resultados.append([])

hilos=[]
for i in range(num_hilos):
    hilo=threading.Thread(target=ordenar_parte,args=(sublistas[i],i))
    hilos.append(hilo)
    hilo.start()
```

```
26
27 for hilo in hilos:
28     hilo.join()
29
30 def fusionar_listas(listas):
31     from heapq import merge
32     resultado=listas[0]
33     for lista in listas[1:]:
34         resultado=list(merge(resultado,lista))
35     return resultado
36 #print(lista_ordenada)
37 lista_ordenada = fusionar_listas(resultados)
38 print("Lista ordenada:",lista_ordenada == sorted(lista_original))
```

4.5.3. Desarrollo y Resultados:

- **Desarrollo:** Este ejercicio emplea programación multihilo para ordenar una gran lista de números. Se divide la lista en varias partes que se ordenan en paralelo mediante hilos usando sorted(), y luego se fusionan con un algoritmo eficiente (heapq.merge, basado en heaps). No se necesitó Lock, ya que cada hilo escribe en una posición separada de la lista resultados, lo que evita condiciones de carrera.
- **Resultados:**



El programa divide la lista de un millón de números en partes iguales, las ordena en paralelo y luego las fusiona correctamente. El resultado final se compara con el ordenamiento secuencial como verificación: Lista ordenada: True

4.6. Ejercicio de Bloqueo: Actualización de un Contador Global

4.6.1. Análisis del ejercicio

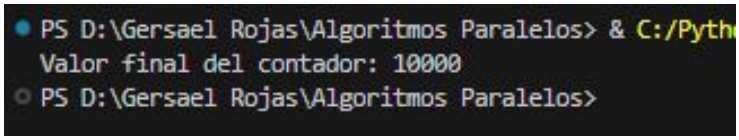
Este ejercicio tiene como objetivo crear un programa en Python que utilice 100 hilos para incrementar un contador global 10,000 veces de manera segura. Cada hilo incrementa el contador 100 veces, y para evitar condiciones de carrera, se debe emplear un mecanismo de sincronización utilizando un Lock. Esto asegura que solo un hilo pueda modificar el contador a la vez, garantizando resultados correctos y evitando errores de actualización concurrente.

4.6.2. Codificacion:

```python
import threading

contador=0
lock=threading.Lock()

def incrementar():
    global contador
    for _ in range(100):
        with lock:
            contador+=1

hilos=[]
for _ in range(100):
    hilo=threading.Thread(target=incrementar)
    hilos.append(hilo)
    hilo.start()

for hilo in hilos:
    hilo.join()

print("Valor final del contador:",contador)
```

4.6.3. Desarrollo y Explicación:

- **Desarollo:** En este ejercicio, el objetivo es realizar 10,000 actualizaciones seguras a un contador global utilizando 100 hilos, cada uno realizando 100 incrementos.
- **Resultados:**



Se han creado 100 hilos, cada uno incrementando un contador global 100 veces.

El uso de Lock asegura que no existan condiciones de carrera al modificar el valor compartido. Resultado esperado del contador: Valor final del contador: 10000

4.7. Ejercicio de Programación en Memoria Compartida:Cálculo de Productos

4.7.1. Análisis del ejercicio

Este ejercicio propone crear un programa en Python que utilice dos hilos para calcular el producto de los elementos de dos listas de números. Cada hilo se encargará de calcular el

producto de la mitad de los elementos en las listas correspondientes, y luego se combinarán los resultados. Para manejar el acceso concurrente a las variables y evitar condiciones de carrera, se debe emplear un Lock. Este ejercicio permite practicar la paralelización de cálculos y la correcta sincronización de hilos al trabajar con datos compartidos.

4.7.2. Codificacion:

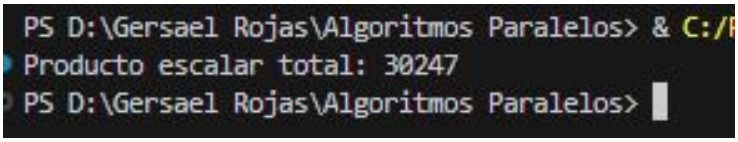
```
import threading
import random

lista1=[random.randint(1, 10) for _ in range(1000)]
lista2=[random.randint(1, 10) for _ in range(1000)]

producto_total=0
lock=threading.Lock()

def producto_parcial(inicio,fin):
    global producto_total
    producto=0
    for i in range(inicio,fin):
        producto+=lista1[i]*lista2[i]
    with lock:
        producto_total+=producto

medio=len(lista1)//2
hilo1=threading.Thread(target=producto_parcial,args=(0,medio))
hilo2=threading.Thread(target=producto_parcial,args=(medio,len(lista1)))

hilo1.start()
hilo2.start()
hilo1.join()
hilo2.join()

print("Producto escalar total:",producto_total)
```

4.7.3. Desarrollo y Explicación:

- **Desarrollo:** En el ejercicio usamos dos listas de 1000 elementos cada una. Cada hilo calcula el producto escalar parcial (es decir, la suma de lista1[i]*lista2[i]). Se emplea un bloqueo (Lock) para asegurar el acceso seguro a la variable compartida ”producto-total”.

- **Resultados:**

  Se generan dos listas aleatorias de 1000 números, y dos hilos calculan la suma de los productos de cada elemento. Se asegura que el resultado final sea correcto gracias al uso del Lock, evitando condiciones de carrera. Resultado esperado (puede variar):



Producto escalar total: 30247 (este valor es solo un ejemplo, ya que las listas son aleatorias).

4.8. Ejercicio de Sincronización Avanzada: Fila de Tareas

4.8.1. Análisis del ejercicio

Este ejercicio consiste en crear un programa en Python donde varios hilos procesen tareas de una cola compartida. Los hilos tomarán las tareas de la cola y las procesarán de manera concurrente. Se debe usar la clase Queue para gestionar la cola de tareas y asegurar que los hilos no accedan simultáneamente a las mismas tareas, utilizando un Lock para sincronización. Este ejercicio permite trabajar con la comunicación entre hilos y la coordinación en el acceso a recursos compartidos, fundamental para evitar condiciones de carrera y garantizar un procesamiento seguro.

4.8.2. Codificacion:

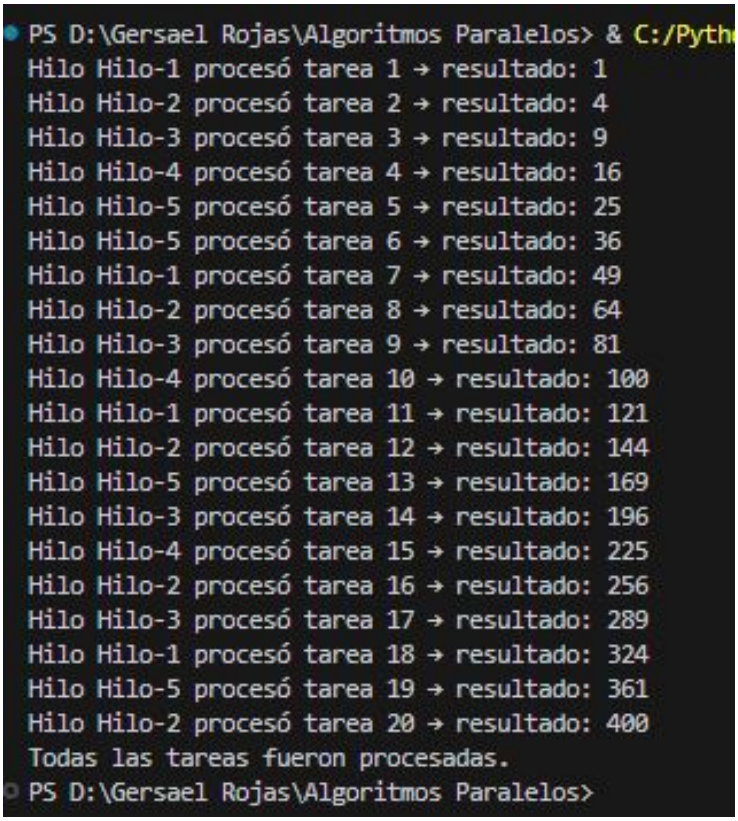
```
import threading
import queue
import time
import random

tareas=queue.Queue()
for i in range(1,21):
    tareas.put(i)

lock=threading.Lock()

def procesar_tarea():
    while not tareas.empty():
        try:
            tarea=tareas.get(timeout=1)
        except queue.Empty:
            break
        resultado=tarea**2
        with lock:
            print(f"Hilo {threading.current_thread().name} proces
   tarea {tarea}     resultado: {resultado}")
        tareas.task_done()
        time.sleep(random.uniform(0.1,0.3))

hilos=[]
```

```python
for i in range(5):
    hilo=threading.Thread(target=procesar_tarea,name=f"Hilo-{i+1}")
    hilos.append(hilo)
    hilo.start()

for hilo in hilos:
    hilo.join()

print("Todas las tareas fueron procesadas.")
```

4.8.3. Desarrollo y Explicación:

- **Desarrollo:** Este ejercicio implementa un sistema donde varios hilos trabajan en paralelo para procesar tareas extraídas de una cola compartida (Queue). Se usa queue.Queue() como una estructura segura para múltiples hilos. Cada hilo ejecuta una función que toma una tarea de la cola y la procesa. Se utiliza un Lock para evitar que varios hilos escriban al mismo tiempo en la consola (o en un registro compartido). Se simula una tarea elevando un número al cuadrado y agregando una pausa aleatoria.

- **Resultados:**



La salida demuestra que cada hilo accedió de forma sincronizada a la cola y al print. Todas las tareas fueron procesadas exactamente una vez. No hubo errores de concurrencia ni resultados duplicados.

4.9. Ejercicio de Programación en Memoria Compartida: Suma y Promedio en Paralelo

4.9.1. Análisis del ejercicio

Este ejercicio tiene como objetivo crear un programa en Python que calcule la suma y el promedio de los elementos de una lista de números utilizando dos hilos en paralelo. Un hilo se encargará de calcular la suma de los números, mientras que el otro calculará el promedio. Para garantizar que los resultados sean precisos y evitar condiciones de carrera, se debe usar un mecanismo de sincronización (como Lock). Esto permitirá gestionar el acceso concurrente a las variables y asegurar que ambos hilos trabajen de manera segura.

4.9.2. Codificacion:

```
    import threading

numeros=[i for i in range(1,1001)]

suma_total=0
promedio_total=0.0
lock=threading.Lock()

def calcular_suma():
    global suma_total
    suma=sum(numeros)
    with lock:
        suma_total=suma

def calcular_promedio():
    global promedio_total
    promedio=sum(numeros)/len(numeros)
    with lock:
        promedio_total=promedio

hilo_suma=threading.Thread(target=calcular_suma)
hilo_promedio=threading.Thread(target=calcular_promedio)

hilo_suma.start()
hilo_promedio.start()

hilo_suma.join()
hilo_promedio.join()

```

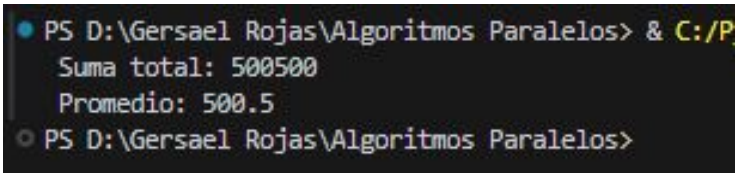
```
30 print(f" Suma total: {suma_total}")
31 print(f" Promedio: {promedio_total}")
```

4.9.3. Desarrollo y Resultados:

- **Desarollo:** Este programa fue diseñado para calcular la suma y el promedio de una lista de números utilizando dos hilos que trabajan en paralelo. La lista contiene los números del 1 al 1000, y tanto la suma como el promedio se calculan utilizando funciones independientes asignadas a distintos hilos. El uso de hilos permite realizar ambos cálculos simultáneamente, lo cual es útil para mejorar el rendimiento en programas con múltiples operaciones.

  Se usó la biblioteca threading para crear los hilos, y una variable Lock para asegurar que las variables compartidas (suma-total y promedio-total) no sean modificadas al mismo tiempo por ambos hilos, evitando así condiciones de carrera. El acceso sincronizado a estas variables garantiza que los valores finales sean correctos y consistentes.

- **Resultado:**



  El resultado de la ejecución del programa muestra correctamente los valores calculados por ambos hilos. Para la lista de números del 1 al 1000, la suma total es 500500, y el promedio es 500.5. Estos resultados son correctos matemáticamente, ya que, la suma de los primeros 1000 números naturales se puede comprobar con la fórmula n(n+1)/2, que da 1000*1001/2 = 500500 y el promedio es la suma dividida entre la cantidad de elementos: 500500 / 1000 = 500.5.

4.10. Ejercicio de Creación de Hilos Dinámicos: Suma de Matrices

4.10.1. Análisis del ejercicio

Este ejercicio propone implementar un programa en Python para realizar la suma de dos matrices de tamaño 1000x1000 utilizando hilos. Cada hilo se encargará de sumar una fila de las matrices correspondientes y luego almacenar el resultado en una matriz final. Para asegurar que las filas se sumen correctamente y evitar condiciones de carrera, se deben utilizar mecanismos de sincronización como Lock. Este enfoque permite dividir el trabajo entre múltiples hilos, mejorando la eficiencia y facilitando el uso de procesamiento paralelo, mientras se asegura la correcta manipulación de los datos compartidos.

4.10.2. Codificacion:

```python
import threading
import random

filas=1000
columnas=1000
matriz1=[[random.randint(1,100) for _ in range(columnas)] for _ in range
    (filas)]
matriz2=[[random.randint(1,100) for _ in range(columnas)] for _ in range
    (filas)]
matriz_resultado = [[0 for _ in range(columnas)] for _ in range(filas)]
lock=threading.Lock()

def sumar_fila(i):
    fila_resultado=[]
    for j in range(columnas):
        fila_resultado.append(matriz1[i][j]+matriz2[i][j])
    with lock:
        matriz_resultado[i]=fila_resultado

hilos = []
for i in range(filas):
    hilo=threading.Thread(target=sumar_fila,args=(i,))
    hilos.append(hilo)
    hilo.start()

for hilo in hilos:
    hilo.join()

print("Fila 0 de la matriz resultado:")
print(matriz_resultado[0])
```

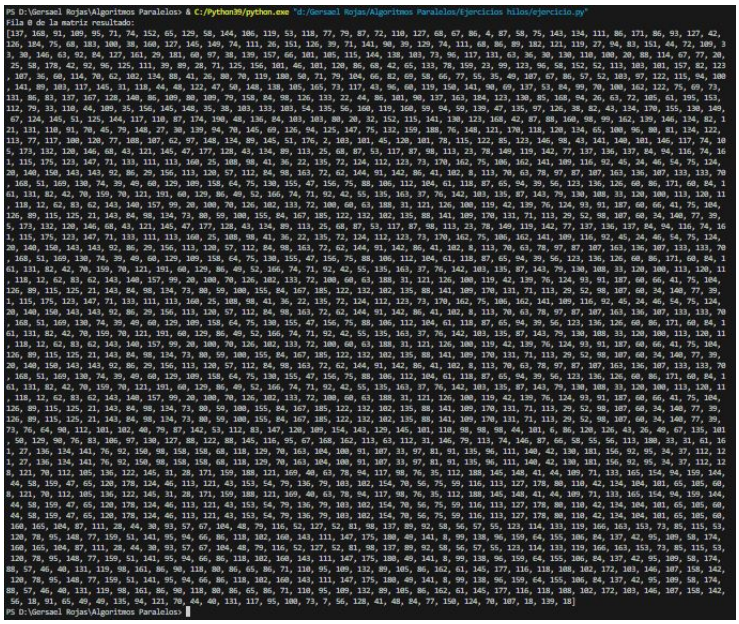
4.10.3. Desarrollo y Resultados:

- **Desarollo:** El programa tiene como objetivo realizar la suma de dos matrices grandes (1000x1000 elementos) utilizando hilos dinámicos. Cada hilo se encarga de calcular la suma de una única fila de ambas matrices, y el resultado se almacena en una tercera matriz final. Este enfoque divide el trabajo eficientemente entre múltiples hilos, maximizando el uso de la capacidad del procesador en sistemas multinúcleo.

  Se utilizaron estructuras como Thread del módulo threading para crear los hilos, y se implementó un Lock para evitar que múltiples hilos modifiquen la misma parte de la matriz resultado al mismo tiempo. Aunque en este caso cada hilo trabaja con una fila distinta (por lo que no habría interferencias directas), el uso de Lock asegura que el acceso a matriz-resultado se realice de forma segura y sin condiciones de carrera, especialmente si se realiza escritura simultánea.

Las matrices se llenan con números aleatorios para simular un caso de prueba más realista y exigente en cuanto a carga de datos. El programa crea un hilo por fila, totalizando 1000 hilos activos que ejecutan en paralelo la operación de suma para cada fila.

- **Resultado:**



Al finalizar la ejecución del programa, se imprime la primera fila de la matriz resultado para verificar que las operaciones se realizaron correctamente. Esta salida muestra que los valores de la fila 0 son la suma elemento por elemento de las filas 0 de las dos matrices originales.

Además, se puede validar que todas las filas fueron correctamente sumadas observando que el programa no presenta errores de sincronización ni condiciones de carrera. Aunque lanzar 1000 hilos simultáneamente no es lo más eficiente en la práctica (debido al contexto de cambio de hilos y el uso de memoria), este enfoque demuestra el uso correcto de hilos dinámicos en la programación concurrente y el manejo adecuado de memoria compartida.

Si se quisiera optimizar, una mejora futura sería usar un ThreadPoolExecutor o limitar la cantidad de hilos activos simultáneamente para adaptarse mejor a la capacidad del hardware.

5. Conclusiones

1. **Coclusion numero uno :** La programación concurrente mejora significativamente el rendimiento al dividir tareas en múltiples hilos, especialmente en operaciones costosas como el ordenamiento de grandes listas o cálculos matemáticos intensivos. Esta técnica permite aprovechar mejor los recursos del procesador.

2. **Coclusion numero dos :** El uso adecuado de mecanismos de sincronización como Lock es esencial para evitar condiciones de carrera, garantizando que los datos compartidos entre hilos se gestionen de forma segura y sin errores, como se demostró en los ejercicios del contador global y del producto escalar.

3. **Coclusion numero tres :** El uso adecuado de mecanismos de sincronización como Lock es esencial para evitar condiciones de carrera, garantizando que los datos compartidos entre hilos se gestionen de forma segura y sin errores, como se demostró en los ejercicios del contador global y del producto escalar.

Repositorio de Github