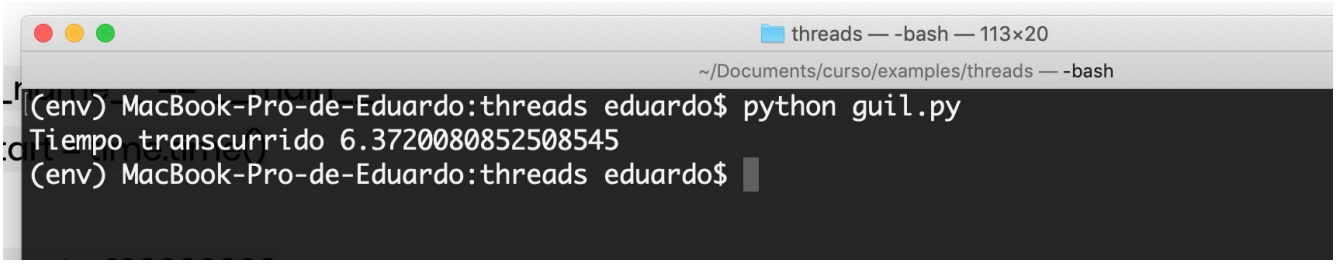
¿Qué es el GIL en Python?

En la actualidad la mayoría de los lenguajes de programación soportan la programación concurrente y la programación en paralelo, pudiendo así ejecutar diferentes tareas en diferentes procesadores, sin embargo, con Python esto no es así. 😟. Con Python al utilizar Threads nunca seremos capaces de lograr un verdadero paralelismo, ya que el lenguaje está diseñado para que un thread y solo un thread pueda ejecutarse a la vez.

Veamos un ejemplo. En este caso se ha definido una función que lo unico que hace es decrecer su parametro(number) hasta llegar a 0.

```
import time


def countdown(number):
    while number > 0:
        number -=1


if __name__ == '__main__':
    start = time.time()


    count = 100000000
    countdown(count)


    print(f'Tiempo transcurrido {time.time() - start }')
```

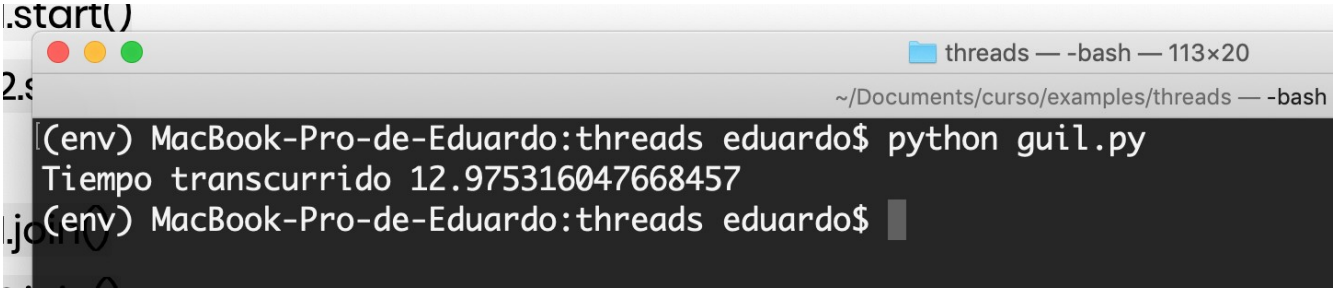
Si se ejecuta el script, tal y como se encuentra ahora, con un solo thread y de forma secuencial, le llevará un apróximadamente 6 segundos finalizar.



Ahora, haremos los mismo pero con dos threads. En teoria, al ejecutarse ambos threads de forma concurrente al script le debería tomar poco más de 6 segundos finalizar, quizás, unos milisegundos más, en teoria. 😥

```
import time
import threading


def countdown(number):
    while number > 0:
        number -=1


if __name__ == '__main__':
    start = time.time()


    count = 100000000


    t1 = threading.Thread(target=countdown, args=(count,))
    t2 = threading.Thread(target=countdown, args=(count,))


    t1.start()
    t2.start()


    t1.join()
    t2.join()


    print(f'Tiempo transcurrido {time.time() - start }')
```

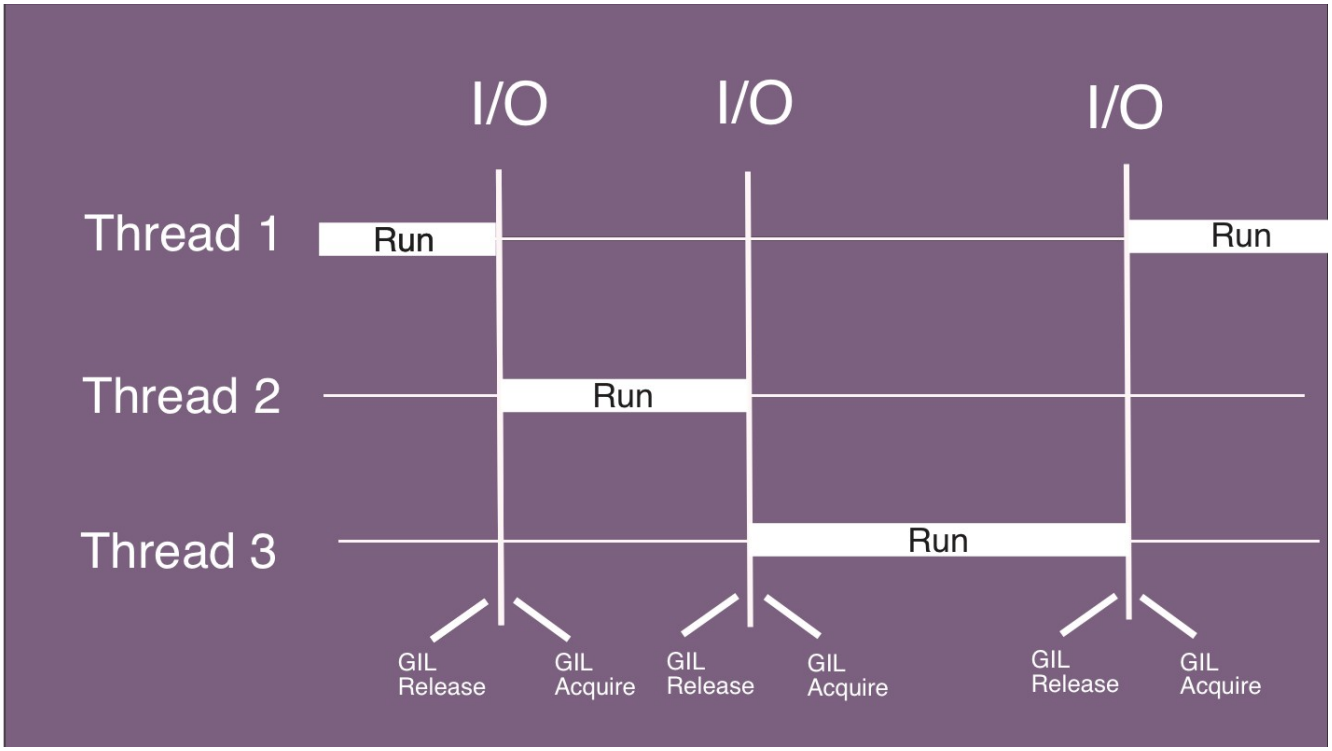
Al escript, de forma concurrente le lleva 12 segundos finalizar, sí, el doble de tiempo apesar de utilizar threads. 😮 Esto gracias al GIL.



Seguro que no te lo esperabas, de hecho yo tampoco, siendo Python uno de los lenguajes más populares y utilizados en todos el mundo como es posible que el mismo lenguaje limite su potencial, pues bien esto se debe a gracias a GIL Global interepter Lock, el villano en todo esto, o quizás no, veamos, por que es tan importante GIL en Python.

GIL

El GIL ,Global interepter Lock de Python, permite que sólo un thread tome el control del intérprete, es decir, que solo un thread puede estar en ejecución a la vez. Esto rara vez tiene repercusiones para quienes desarrollamos programas utilizando solo el thread principal, es más, es probable que nunca nos hayamos dado cuenta, pero, para aquellas personas que desarrollan de forma concurrente este si puede llegar a ser un dolor de cabeza.



Pero, ¿Si GIL genera cuellos de botella, por qué se introdujo en primera instancia y por qué simplemente no se quita? Muy buenas preguntas, y para responderlas es necesario comprender como funciona Python internamente.

En términos simples Python posee un concepto llamado conteo de referencias, el cual le permite conocer al intérprete cuando un objeto está siendo utilizando y cuando no. Es algo bastante simple.

Por ejemplo, si mi variable *A*, posee referencias, por lo menos una o más, entonces, se concluye que la variable está siendo utilizada en alguna parte del programa. Por otro lado, si la variable no posee referencia, se concluye que no se está utilizando, y es allí donde entra el recolector de basura y libera la memoria.

Veamos un ejemplo.

```
import sys


A = 'Hola, soy una referencia'


sys.getrefcount(A)
>>> 2
```

En este casa obtenemos como resultado dos, ya que la variable A cuenta con dos referencias, la asignación y en el llamado a la función.

Bastante sencillo ¿no?

Pues bien, el trabajo de GIL es impedir que múltiples threads decrementen la referencia de algún objeto mientras otros están haciendo uso de ella. Como la naturaleza de un thread es trabajar de forma concurrente, en teoría, es posible que un thread le indique al intérprete que una variable se ha dejado de utilizar, cuando realmente otro thread aun sigue trabajando con ella.

Para evitar este problema se implementó GIL, permitiendo que sólo un thread tome el control del intérprete.

GIL no solo resolvio el problema del conteo de referencias, también hizo que la implementación de Python sea mucho más sencilla, a la vez que incrementa la velocidad de ejecución cuando trabajamos con un único thread.

En palabras de Larry Hastings: *la decisión del diseño de GIL es una de las cosas que hizo que Python fuera tan popular como lo es hoy*.

Ahora, ya sabemos que GIL previene que espacios en memoria sean liberados cuando aún están siendo utilizados, ok, pero, ¿no hay forma en crear algún otro mecanismo en el recolector de basura para evitar el problema? pues, dejame

decirte que Guido van Rossum ,creador de Python dice que GIL esta aquí para quedarse, esto lo explica muy bien en un post al cual te invito le heches un ojo.

Pero en esencia, eliminar GIL traería muchos más problemas que ventajas. Programas que ya se encuentran en producción podrían dejar de comportarse como lo hacen ahora para simplemente fallar. Así que, mejor no. (Yo puedo vivir con GIL)

Multiprocesamientos

Pero no te preocupes, no todo está perdido. Si realmente queremos que las tareas se ejecuten de forma paralela debemos optar por el multiprocesamiento sobre el multithreading; de esta forma cada proceso tendrá su propio intérprete y podrá ejecutarse de manera independiente, logrando así evitar el cuello de botella de GIL y aprovechando todo el potencial de nuestros equipos.

La mejor forma de trabajar procesos en Python es sin duda con la librería multiprocessing.

Aquí un pequeño ejemplo de como crear nuestros propios Procesos. Haz la prueba por ti mismo y verás el resultado.

```
import time
import threading
import multiprocessing


def countdown(number):
    while number > 0:
        number -=1


if __name__ == '__main__':
    start = time.time()


    count = 100000000


    t1 = multiprocessing.Process(target=countdown, args=(count,))
    t2 = multiprocessing.Process(target=countdown, args=(count,))


    t1.start()
    t2.start()


    t1.join()
    t2.join()


    print(f'Tiempo transcurrido {time.time() - start }')
```