THREADING

Juan Pablo Sánchez - Alonso Gac

Qué hemos visto hasta ahora?

Programas que ejecutan sólo una secuencia de instrucciones a la vez

- Siguen un único flujo que comienza, ejecuta instrucciones, y en algún momento termina.
- Muchas aplicaciones realizan múltiples acciones simultáneamente.
- Programas que ejecutan una secuencia de instrucciones a la vez no permiten implementar este tipo de comportamiento.

Qué es un thread?

Un thread es una secuencia de instrucciones que puede ser ejecutada en paralelo con otras, lo que permite realizar más de una acción a la vez.

Ejemplos de uso de threads

- Juegos.
- Cálculos o funciones que utilicen muchos recursos y deban mantener su interfaz funcionando.
- Funciones independientes.

Cómo se usan los threads?

Para utilizar threads debemos comenzar por importar la librería correspondiente

```
import threading
```

Métodos de Thread

- `start()`
- `join()`
- `is_alive()`

Cómo se usan los threads?

La clase `Thread` representa un hilo (o thread). Cada thread ejecutará una secuencia de instrucciones de manera simultánea al resto del programa. Los threads reciben la función por ejecutar en el parámetro `target`.

```
hilo = threading.Thread(target=mi_funcion,
 args=[arg1,arg2,...,arg], name="Mi hilo")
```

Cómo se usan los threads?

Al crear una instancia de `Thread` este NO se ejecuta automáticamente. Para efectivamente ejecutar el thread, se debe llamar a la función `start()`

```
hilo.start()
```

Veamos un ejemplo

`ejemplo_threading_0.py`

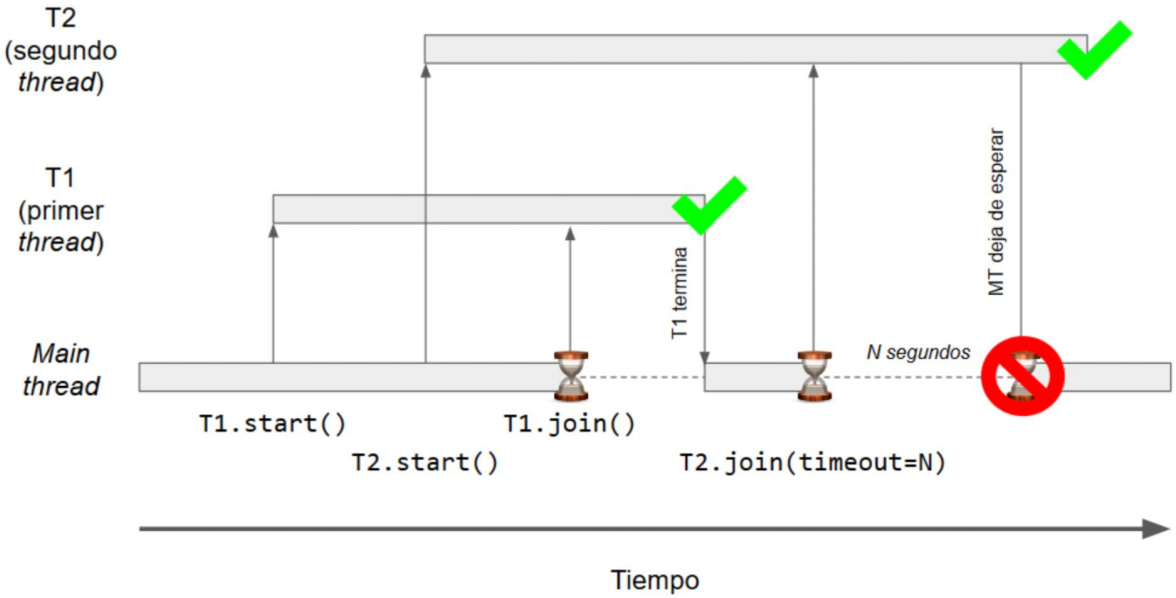
Y si necesito más de un thread?

Es posible usar más de un thread, todo lo que necesitamos es darle un nombre único a la variable que lo contenga.

Veamos un ejemplo

`ejemplo_threading_1.py`

Método `join()`


T2
(segundo
thread)
T1
(primer
thread)
Main
thread
T1 termina
MT deja de esperar
N segundos
T1.start()
T1.join()
T2.start()
T2.join(timeout=N)
Tiempo

Veamos un ejemplo

`ejemplo_threading_2.py`

Método `is_alive()`

Este método nos permite identificar si un thread todavía está en funcionamiento. Retorna `True` si sigue en funcionamiento y `False` si es que no.

Clases como threads

Podemos definir nuestras propias clases que hereden de `Thread` para representar threads con un comportamiento en común. Debemos definir el método `run()` el cual se ejecuta al llamar al método `start()`.

Clases como threads

```
class MiThread(threading.Thread):
    def __init__(self, argumentos):
        super().__init__()
        ...
    def run(self):
        ...
```

Veamos un ejemplo

`ejemplo_threading_3.py`

Daemon threads

Hasta ahora, el programa principal espera a que todos los threads terminen antes de terminar su ejecución. Los daemon threads son aquellos que, a pesar que estén en ejecución, no impiden que el programa principal termine.

Daemon threads

```
# Opción 1

hilo = threading.Thread(target=funcion, daemon=True)

# Opción 2

hilo_2 = threading.Thread(target=funcion)
hilo_2.daemon = True
```

Timers

La clase `Timer` es una subclase de la clase `Thread`. Permite ejecutar un proceso después que ha pasado un tiempo determinado. El método `cancel()` permite cancelar la ejecución del timer antes que este sea ejecutado.

```
t1 = threading.Timer(10.0, funcion)
t1.start() # comenzará después de 10 segundos
```

¿Qué ocurre si dos threads modifican una misma variable?

Cuando dos threads modifican una misma variable durante su ejecución, se produce una condición de carrera o *Race condition.*

Para evitar esto, utilizamos mecanismos de sincronización como *Locks* y *Eventos.*

Locks

Un lock nos permite limitar el acceso a una variable por parte de los threads, de esta forma se obtiene el comportamiento esperado.

Para utilizar un lock, comenzamos por crear una instancia.

```
lock_global = threading.Lock()
```

Veamos un ejemplo

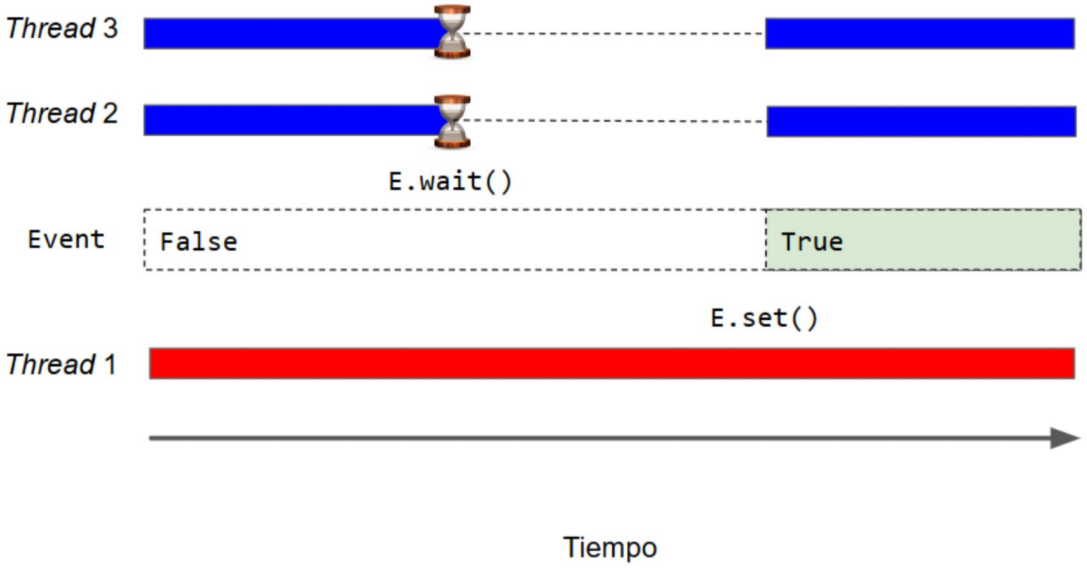
`ejemplo_threading_4.py`

Señales entre threads (eventos)

En ciertas ocasiones vamos a necesitar que un thread espere que ocurra un evento para continuar con sus operaciones. Para ello existen los objetos `Event`, donde un thread emite una señal y otros esperan dicha señal.

Un thread puede esperar una señal llamando al método `wait()` de `Event`. Para activar la señal, un thread debe llamar al método `set()`. Finalmente, podemos resetear una señal llamando a `clear()` de `Event`.

Señales entre threads (eventos)



Veamos un ejemplo

`ejemplo_threading_5.py`

Ahora, una actividad!

Realizaremos una pequeña actividad para poner en práctica todo lo que vimos 🎉

`actividad_threading.py`

¡Muchas gracias!