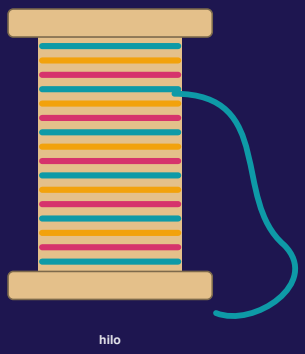


IIC2233 • PROGRAMACIÓN AVANZADA

THREADING

Hilos, concurrencia y sincronización

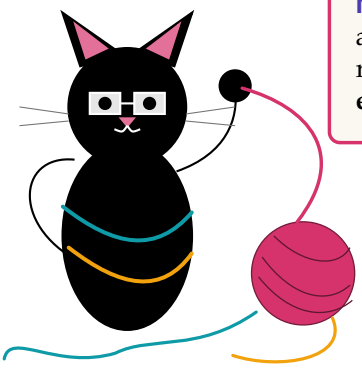


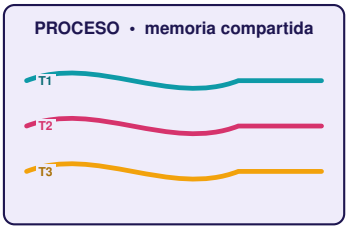
`Thread` • `start` • `join` • `daemon` • `Lock` • `Event` • `Queue`

Felipín quiso tejer una bufanda. Lanzó tres hilos de trabajo que tiraban *del mismo ovillo* sin coordinarse: uno medía el largo justo cuando otro lo estaba cortando, dos esperaban que el otro soltara la aguja, y el programa principal nunca supo cuándo terminaron. Resultado: una bufanda hecha un nudo y un gato hecho un ovillo.

No seas como Felipín. En este resumen aprenderás a crear **threads**, a esperarlos con `join`, a proteger recursos compartidos con **locks**, a coordinarlos con **eventos y colas**, y a evitar los temidos **deadlocks**.







`Thread` → `start()` → `join()` → `Lock` → `Event` → `Queue`



Resumen de Contenidos

IIC2233 — Programación Avanzada • Edición Hilos

Resumen elaborado por el equipo de ayudantía, en base al material oficial del curso

Tabla de Contenidos

1 Introducción 2

2 Procesos, núcleos y concurrencia 3

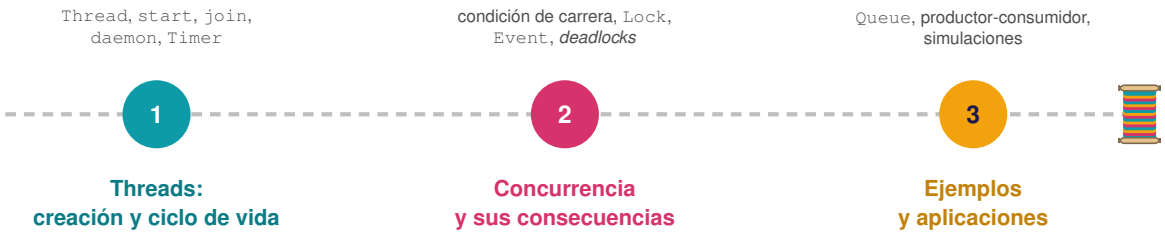
3 Threads en Python 5

4 Sincronización 14

5 *Deadlocks* 20

6 Ejemplos y aplicaciones 23

7 Hoja de referencia 29

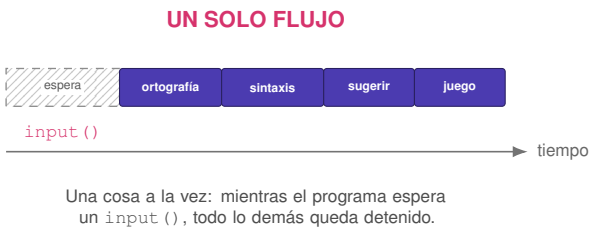


1 Introducción

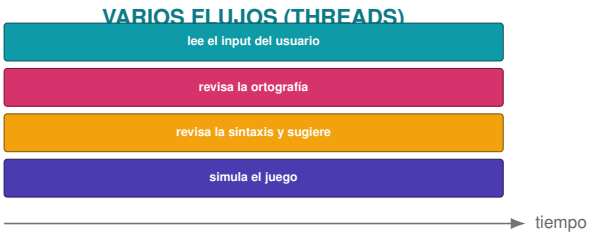
Hasta ahora, todos los programas que hemos escrito siguen **un único flujo de instrucciones**: comienzan, ejecutan una instrucción tras otra y en algún momento terminan. Eso tiene una consecuencia incómoda: nuestro programa solo puede estar atento a **una cosa a la vez**. Cuando ejecutamos `input()`, el programa se detiene esperando la respuesta, y si el usuario aprieta una tecla mientras el programa está haciendo otra cosa, no tiene cómo enterarse.

Sin embargo, casi todas las aplicaciones que usamos hacen **varias cosas simultáneamente**: un procesador de texto revisa la ortografía mientras escribimos, un IDE colorea la sintaxis y sugiere autocompletados, un videojuego mueve a nuestro personaje mientras el resto del mundo sigue evolucionando. Programar eso con un solo flujo obligaría a armar un ciclo gigante que, por cada `input()`, revise ortografía, sintaxis, sugerencias, simule el juego... Sería un enredo.

La solución es que un mismo programa pueda **seguir varios flujos de instrucciones a la vez**, cada uno encargado de una tarea. Cada uno de esos flujos es un ***thread*** (hilo).



Una cosa a la vez: mientras el programa espera un `input()`, todo lo demás queda detenido.



Cada tarea avanza por su cuenta, dentro del mismo programa.

Los tres bloques de este resumen

- **Threads: creación y ciclo de vida** (Secciones 2 y 3): qué son un proceso, un núcleo y un *thread*; diferencia entre **concurrencia** y **paralelismo**; cómo crear *threads* con `Thread`, esperarlos con `join()`, consultarlos con `is_alive()`, dejarlos correr en segundo plano (*daemons*) y programarlos con `Timer`.
- **Concurrencia y sus consecuencias** (Secciones 4 y 5): qué puede salir mal cuando varios *threads* tocan el

mismo dato, y cómo se resuelve con `Lock` y `Event`; qué es un *deadlock*.

- **Ejemplos y aplicaciones** (Sección 6): *locks* dentro de una subclase de `Thread`, el patrón **productor-consumidor** con `Queue`, y simulaciones con *threads*.

★ Cómo leer este resumen

Cada bloque de código va acompañado de una explicación del tipo *"¿Qué hace este código?"*. Los recuadros **magenta** marcan errores típicos y trampas; los **dorados** son definiciones importantes; los **turquesa** son consejos; los **verdes** muestran la **salida** de un programa; y las **analogías del taller** son metáforas para fijar las ideas.
En los diagramas de tiempo usamos siempre los mismos colores: el *thread* principal en **índigo**, y los hilos **T1 (turquesa)**, **T2 (magenta)** y **T3 (ámbar)**. Un bloque **rayado** significa que ese *thread* está **bloqueado esperando** algo.
Importante: los ejemplos con *threads* **no son deterministas**: el orden exacto en que se intercalan los mensajes puede cambiar de una ejecución a otra. Además, el material del curso recomienda ejecutar todos los ejemplos **desde una consola** y no desde un Jupyter Notebook, porque este último puede ocultar efectos como los de *daemon* o `Timer`.

2 Procesos, núcleos y concurrencia

2.1 Procesos y núcleos

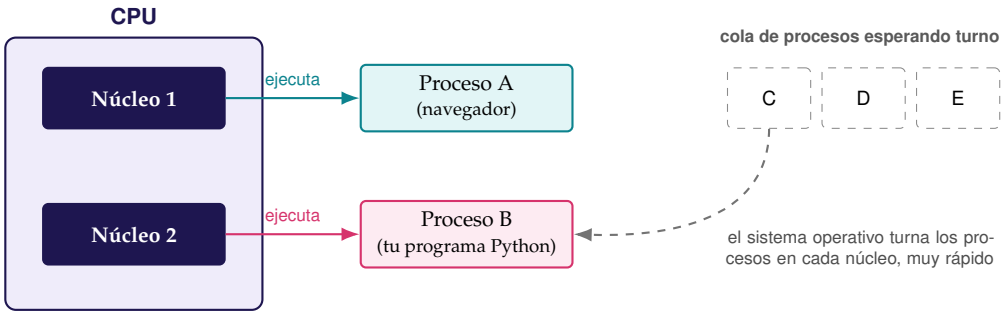
Nuestro computador ejecuta muchos programas al mismo tiempo: un navegador, un reproductor de música, un editor, nuestro programa de Python... Esto lo consigue el sistema operativo (SO) a través de la abstracción de **proceso**.

■ Proceso

Un **proceso** es un **programa en ejecución**. Ocupa un espacio de la memoria (RAM) del computador y ejecuta un flujo de instrucciones, partiendo desde la primera línea de código.

Las instrucciones las ejecuta la **CPU**, que posee uno o más **núcleos** (*cores*). Un núcleo lee una instrucción, la ejecuta y pasa a la siguiente, hasta que se acaban. Cuando corre nuestro programa Python, hay un núcleo siguiendo ese flujo de instrucciones.

Los sistemas modernos tienen **múltiples núcleos** (*dualcore*, *quadcore*, *octacore*...), incluso los teléfonos y los televisores. Con varios núcleos puede haber un proceso en cada núcleo, ejecutándose de forma **realmente simultánea**. Cuando hay **más procesos que núcleos**, los procesos esperan en una cola y el SO se encarga de darle a cada uno un ratito de núcleo, pausarlo y pasar a otro. Esto ocurre tan rápido que parece que todos avanzan a la vez.



★ Analogía del taller

Imagina un **taller de costura**. Cada **núcleo** es una **máquina de coser**: solo puede hacer una puntada a la vez. Cada **proceso** es un **encargo** distinto (un vestido, unas cortinas). Si el taller tiene dos máquinas, dos encargos avanzan de verdad al mismo tiempo; si hay cinco encargos, el jefe del taller (el SO) va cambiando qué encargo pasa por cada máquina.

2.2 Paralelismo vs. concurrencia

Son dos conceptos que se confunden a menudo, y conviene tenerlos claros antes de hablar de *threads*.

■ Paralelismo

Ejecución **simultánea** de varias tareas en **distintos núcleos o procesadores**: operan *al mismo tiempo* en hardware separado. Por eso **no existe paralelismo real con un solo núcleo**.

■ Concurrencia

Manejo de varias tareas de forma **intercalada** en un mismo núcleo. En cada instante solo una tarea se ejecuta, pero los **cambios de contexto** son tan rápidos que *parece* simultáneo.



	Concurrencia	Paralelismo
Núcleos necesarios	Basta con **uno**.	Se necesitan **dos o más**.
En un instante dado	Solo una tarea corre; las otras esperan su turno.	Varias tareas corren realmente a la vez.
Idea central	**Gestionar** varias tareas, intercalándolas.	**Ejecutar** varias tareas al mismo tiempo.
En Python	Módulo `threading`.	`multiprocessing` (o desactivar el GIL en versiones recientes).

★ Analogía del taller

Concurrencia: una sola costurera con una sola máquina que alterna entre dos vestidos: cose un poco de uno, un poco del otro. **Paralelismo:** dos costureras, cada una con su máquina, cosiendo dos vestidos de verdad al mismo tiempo.

2.3 El GIL de Python

En Python, la concurrencia se logra con *threads* mediante el módulo `threading`. Pero Python tiene un mecanismo llamado ***Global Interpreter Lock*** (**GIL**), que **impide ejecutar *threads* en paralelo**, sin importar cuántos núcleos tenga el computador.

■ GIL: *Global Interpreter Lock*

Es un *lock* global del intérprete que se asegura de que **un solo *thread*** pueda ejecutar código Python en un instante dado. Por lo tanto, los *threads* de Python son **concurrentes, pero no paralelos**. Para lograr paralelismo real en tareas que usan mucha CPU se puede usar la librería `multiprocessing` o, en versiones recientes de Python, desactivar manualmente el GIL.

★ Entonces, ¿para qué sirven los threads en Python?

Aunque no aceleren el cómputo puro, siguen siendo muy útiles: permiten **organizar el programa** en partes independientes (sin programar a mano el salto entre funcionalidades) y sirven cuando el programa pasa mucho tiempo **esperando** (el usuario, un archivo, la red, un `sleep`), porque mientras un *thread* espera, otro puede avanzar. Es la razón de que en los ejemplos de este resumen usemos `time.sleep()` para simular trabajo.

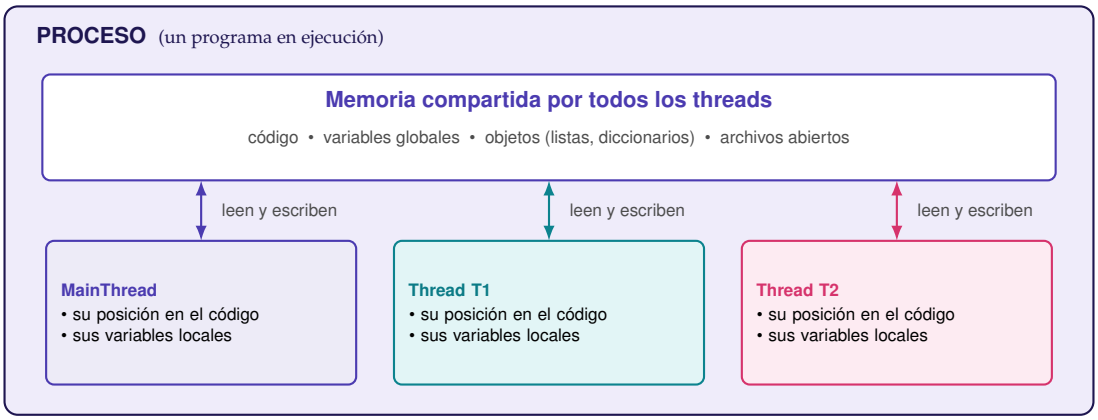
3 Threads en Python

3.1 ¿Qué es un *thread*?

■ Thread (hilo)

Un ***thread*** es una **unidad de ejecución de código dentro de un proceso**. Si el proceso es el programa cargado en memoria, con su código y sus variables, el *thread* es quien **lleva la cuenta de en qué parte del código estamos ejecutando**.

Todo proceso, al crearse, posee **al menos un *thread***: el ***thread* principal** (`MainThread`), que ejecuta el flujo principal del programa. Desde ahí podemos crear más *threads* (y un *thread* también puede crear otros), de modo que el programa ejecute varias partes del código a la vez. Cada *thread* lleva **su propio registro** de variables locales y de la posición en que va dentro del código, pero **todos comparten la memoria del proceso**: las variables globales, los objetos, los archivos abiertos.

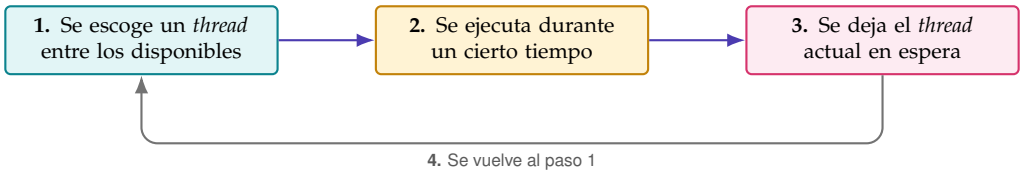


★ Analogía del taller

El **taller** completo es el proceso: tiene una mesa de trabajo, telas, tijeras, hilos (la memoria compartida). Cada **costurera** es un *thread*: lleva su propia cuenta de en qué costura va y qué pieza tiene en las manos (sus variables locales), pero todas trabajan sobre las **mismas mesas y materiales**. De ahí viene tanto el poder de los *threads* como sus problemas: dos costureras pueden intentar usar la misma tela a la vez (Sección 4).

3.2 ¿Quién corre y cuándo? *Thread scheduling*

Gracias al sistema operativo, cada *thread* recibe una porción de tiempo en el núcleo. La decisión de **qué *thread* se ejecuta a continuación y por cuánto tiempo** se conoce como ***thread scheduling*** o ***time slicing***, y funciona (en general) así:



Con un solo núcleo no hay paralelismo real: los turnos pasan tan rápido que parece que todos avanzan a la vez. Por eso, **en un solo núcleo, usar *threads* hace que el programa sea algo más lento**, pues hay que gastar tiempo controlando los turnos. A cambio, la programación es mucho más clara: cada *thread* se hace cargo de un aspecto del programa, y no tenemos que programar a mano el salto entre funcionalidades dentro de un solo flujo.

! No puedes suponer el orden

El SO decide cuándo se pausa un *thread* y cuál sigue. **Nunca** se puede asumir que se ejecutará una instrucción del *thread* 1, luego una del 2, luego una del 1, y así sucesivamente: el intercalado es impredecible y puede cambiar de una ejecución a otra. Casi todos los errores de *threading* nacen de olvidar esto.

3.3 Usos de los *threads*

Uso	Idea y ejemplo
Funciones independientes sobre los mismos datos	Un *thread* recibe texto del usuario y lo deja en una cola; otro lo procesa línea a línea y lo saca de la cola. Este patrón se llama **productor-consumidor** (Sección 6).
Interfaces gráficas	Se atiende al usuario (texto, botones) mientras otros *threads* hacen cómputo pesado o actualizan lo visual.
Aplicaciones multiusuario	Cada *thread* atiende las peticiones de un usuario: por ejemplo, un servidor web que debe responder a varios clientes a la vez.

3.4 Creación de *threads*

En Python 3, los *threads* se manejan con el módulo `threading`, y cada *thread* es un **objeto** de la clase `Thread`.

3.4.1 `Thread`, `target` y `start()`

A la clase `Thread` le entregamos, mediante el parámetro `target`, la **función** que el *thread* debe ejecutar.

```python
import threading

def contar_diez_ovejas() -> None:
    print("Tengo sueño...")
    for numero in range(1, 11):
        print(f"({numero} oveja{'s' if numero > 1 else ''})")
    print("A dormir...")

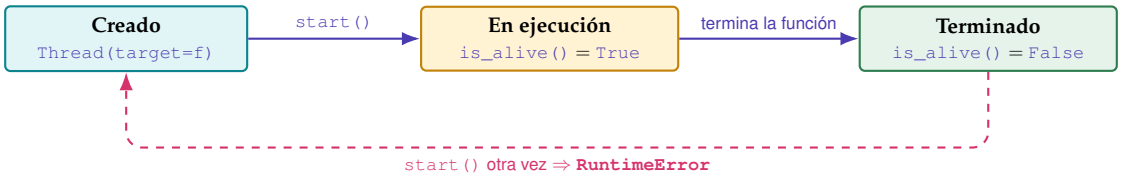
mi_hilo = threading.Thread(target=contar_diez_ovejas)
# Hasta aqui NO se imprime nada: el thread fue creado, pero no ha partido.

mi_hilo.start()      # ahora si: el thread empieza a ejecutar contar_diez_ovejas
```

¿Qué hace este código? Crear la instancia `Thread` **no ejecuta nada**: solo define un hilo que *podría* ejecutar la función. Recién al llamar a `start()` el *thread* comienza a correr `contar_diez_ovejas` de forma concurrente con el resto del programa.

! Un `Thread` es de un solo uso

Una vez ejecutado, un `Thread` **no puede volver a iniciarse**. Si llamamos a `mi_hilo.start()` por segunda vez, Python lanza un `RuntimeError`. Para repetir la tarea hay que **crear una nueva instancia**.



3.4.2 Nombres y `current_thread()`

La clase `Thread` admite el argumento `name`, un nombre para identificar al *thread*. El *thread* principal se llama, por defecto, `"MainThread"`. La función `threading.current_thread()` retorna **una referencia al *thread* que está ejecutando ese código en ese momento**, y así podemos acceder a sus atributos (como `name`).

```python
def saludar() -> None:
    thread_actual = threading.current_thread()
    print(f"Hola desde {thread_actual.name}\n")

hilo_1 = threading.Thread(name="Mi thread 1", target=saludar)
hilo_2 = threading.Thread(name="Mi thread 2", target=saludar)

hilo_1.start()      # saludar() corre en "Mi thread 1"
```

```
hilo_2.start()      # saludar() corre en "Mi thread 2"
saludar()           # saludar() corre en el thread principal
```

Una salida posible (el orden puede variar)

```
Hola desde Mi thread 1
Hola desde Mi thread 2
Hola desde MainThread
```

¿Qué hace este código? La misma función `saludar` se ejecuta en **tres *threads* distintos**, y cada uno imprime su propio nombre, incluido el principal.

3.4.3 Ejemplo: tres *threads* con distinta duración

Creamos tres *threads* que ejecutan funciones que **duermen** distinta cantidad de tiempo (con `time.sleep`) para simular trabajo. Fíjate en que el programa principal **sigue de largo** apenas parten los *threads*.

```
import threading
import time

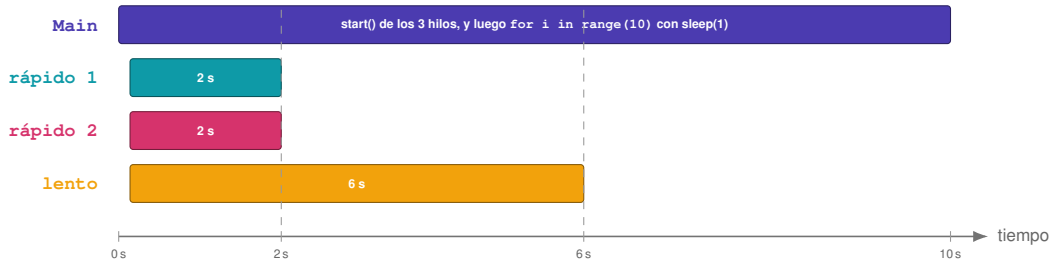
def trabajador_rapido() -> None:          # toma 2 segundos
    nombre = threading.current_thread().name
    print(f"{nombre} partiendo...")
    time.sleep(2)
    print(f"{nombre} terminando...")

def trabajador_lento() -> None:           # toma 6 segundos
    nombre = threading.current_thread().name
    print(f"{nombre} partiendo...")
    time.sleep(6)
    print(f"{nombre} terminando...")

hilo_lento = threading.Thread(name="Hilo lento (6s)", target=trabajador_lento)
hilo_rapido_1 = threading.Thread(name="Hilo rápido (2s)", target=trabajador_rapido)
hilo_rapido_2 = threading.Thread(target=trabajador_rapido)    # nombre por defecto

hilo_rapido_1.start()
hilo_rapido_2.start()
hilo_lento.start()
print("Thread principal: Fueron iniciados 3 threads")

for i in range(10):                       # el principal sigue trabajando
    print(f"Thread principal: Segundo actual: {i}")
    time.sleep(1)
```



¿Qué hace este código? Los tres hilos parten casi a la vez y terminan de forma independiente: los rápidos a los 2 segundos y el lento a los 6. Mientras tanto, el *thread* principal imprime el segundo actual durante 10 segundos. **El programa termina solo cuando todos los *threads* (no *daemon*) han terminado**, aunque el principal ya haya llegado al final de su código.

3.4.4 Pasando argumentos: `args` y `kwargs`

Los argumentos de la función `target` se entregan a través de `args` (una **tupla**) o `kwargs` (un **diccionario**).

```
def contar_ovejas_hasta(max_ovejas: int) -> None:
    nombre = threading.current_thread().name
    print(f"{nombre} tiene sueño...")
    for numero in range(1, max_ovejas + 1):
```

```
        time.sleep(1)
        print(f"({nombre}: {numero} oveja{'s' if numero > 1 else ''})")
    print(f"{nombre} a dormir...")

t1 = threading.Thread(name="Thread 1", target=contar_ovejas_hasta, args=(10,))
t2 = threading.Thread(name="Thread 2", target=contar_ovejas_hasta,
                      kwargs={"max_ovejas": 15})
t1.start()
t2.start()
```

★ La coma de `(10,)`

`args` debe ser una tupla. Con un único argumento hay que escribir la coma: `args=(10,)`. Sin la coma, `(10)` es solo un entero entre paréntesis y Python lanzará un error al intentar desempacarlo.

3.4.5 *Threads* como clases: heredar de `Thread`

En línea con la programación orientada a objetos, también podemos crear *threads* como **instancias de una clase que hereda de `Thread`**. Así agrupamos comportamiento común y estado propio. Para definir qué ejecuta el *thread* hacemos ***override* del método `run()`**, que se ejecuta cuando llamamos a `start()`.

```
class CuentaOvejas(threading.Thread):           # hereda de Thread
    def __init__(self, nombre: str, max_ovejas: int) -> None:
        super().__init__(name=nombre)           # SIEMPRE se llama primero al init original
        self.max_ovejas = max_ovejas            # atributo de instancia extra

    def run(self) -> None:                      # instrucciones del thread
        print(f"\t{self.name} tiene sueño...")
        inicio = time.time()
        for numero in range(1, self.max_ovejas + 1):
            time.sleep(1)
            print(f"\t({self.name}: {numero} oveja{'s' if numero > 1 else ''})")
        print(f"\t{self.name} se durmió después de {time.time() - inicio} seg.")


class CuentaLiebres(threading.Thread):          # las liebres cuentan el doble de rápido
    def __init__(self, nombre: str, max_liebres: int) -> None:
        super().__init__(name=nombre)
        self.max_liebres = max_liebres

    def run(self) -> None:
        for numero in range(1, self.max_liebres + 1):
            if numero % 2 == 1:
                time.sleep(1)                   # duerme 1 s cada dos liebres
            print(f"\t({self.name}: {numero} liebre{'s' if numero > 1 else ''})")


cuenta_ovejas = CuentaOvejas("Gogeta", 10)
cuenta_liebres = CuentaLiebres("Anya", 10)
cuenta_ovejas.start()
cuenta_liebres.start()
```

¿Qué hace este código? Cada clase describe un tipo de *thread* con su propio `run()`. Al hacer `start()`, Python crea el hilo y ejecuta `run()` dentro de él. Como cada objeto es un *thread*, puede guardar **atributos propios** (`max_ovejas`) y usar `self.name`.

! `start()` no es `run()`

Hay que llamar a `start()`, **nunca** a `run()` directamente. `start()` crea el nuevo flujo de ejecución y *ese* flujo llama a `run()`; si llamas a `run()` tú mismo, el código se ejecuta **en el *thread* que lo llamó**, como una función cualquiera, y no hay concurrencia.

	`Thread(target=funcion)`	Subclase con `run()`
Se define	Una función normal.	Una clase que hereda de `Thread` y sobreescribe `run()`.
Estado propio	Hay que pasarlo por `args`/`kwargs`.	Se guarda en atributos (`self.algo`).
Conviene cuando	La tarea es simple y única.	Hay muchos *threads* del mismo tipo o con comportamiento y datos propios.
Recordar	Pasar los argumentos como tupla/diccionario.	Llamar **siempre** a `super().__init__()`.

3.4.6 ¿Qué pasa con los `print`?

A partir de estos ejemplos es probable que, al ejecutar `print`, **falten o sobren saltos de línea**. La razón es que, a nivel de máquina, escribir el texto y escribir el salto de línea son **dos instrucciones distintas**. Un *thread* puede imprimir su texto, ser pausado, y entonces otro *thread* imprime su texto y su salto de línea; al volver el primero, imprime el salto que le faltaba.

■ Ejemplo de salida con líneas mezcladas

```
Thread 1: hola Thread 2: hola
Thread 3: hola
```

★ Un `print` no es atómico

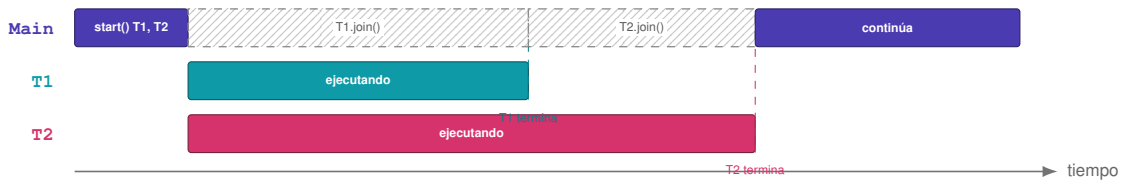
Este comportamiento no es un error de tu código: es una consecuencia de que dos *threads* usan un mismo recurso (la consola) sin coordinarse. Es un adelanto de lo que veremos en la Sección 4.

3.5 Esperando a otros *threads*: `join()`

Hay ocasiones en que el programa principal necesita que otros *threads* terminen **antes** de ejecutar ciertas instrucciones. Por ejemplo, un programa que debe calcular el total de compras de una tienda, cuyo detalle está en cinco archivos con distintos formatos: delega en cinco *threads* la lectura y suma de cada archivo, **espera** a que todos terminen y recién entonces suma los cinco resultados.

Para eso existe el método `join(timeout=None)`, que se llama **después** de `start()`:

- El ***thread* que llama a `join()`** queda **bloqueado** hasta que el *thread* referenciado termine.
- Si se entrega un `timeout` (en segundos), espera **como máximo** ese tiempo y luego continúa, haya terminado o no el otro *thread*.



En el diagrama, es el *Main Thread* quien llama a `T1.join()` y `T2.join()`, así que **es él quien se bloquea**: primero espera a T1 y luego a T2. Pero ojo: **cualquier *thread* puede esperar a cualquier otro** usando `join()`, no solo el principal.

```
naruto = CuentaOvejas("Naruto", 5)       # las clases del ejemplo anterior
goku = CuentaOvejas("Goku", 2)
komi = CuentaLiebres("Komi", 5)
anya = CuentaLiebres("Anya", 7)
chisato = CuentaLiebres("Chisato", 20)

for protagonista in (naruto, goku, komi, anya, chisato):
    protagonista.start()

print("Minions: Los protagonistas se fueron a dormir...")
```

```
goku.join()          # sin timeout: espera lo que sea necesario
print("Minions: ¡GOKU SE DURMIÓ!")
naruto.join()
print("Minions: ¡NARUTO SE DURMIÓ!")
komi.join()
anya.join()
chisato.join(1)      # espera como MÁXIMO 1 segundo mas
print("Minions: ¡(casi todos) se durmieron! ¡A festejar!")
```

¿Qué hace este código? El programa principal (los "*minions*") lanza cinco *threads* y luego va esperándolos con `join()`. No festeja hasta que la mayoría se durmió. Con `chisato.join(1)` solo espera un segundo más: como Chisato cuenta hasta 20, probablemente ya no habrá terminado, y el programa sigue igual.

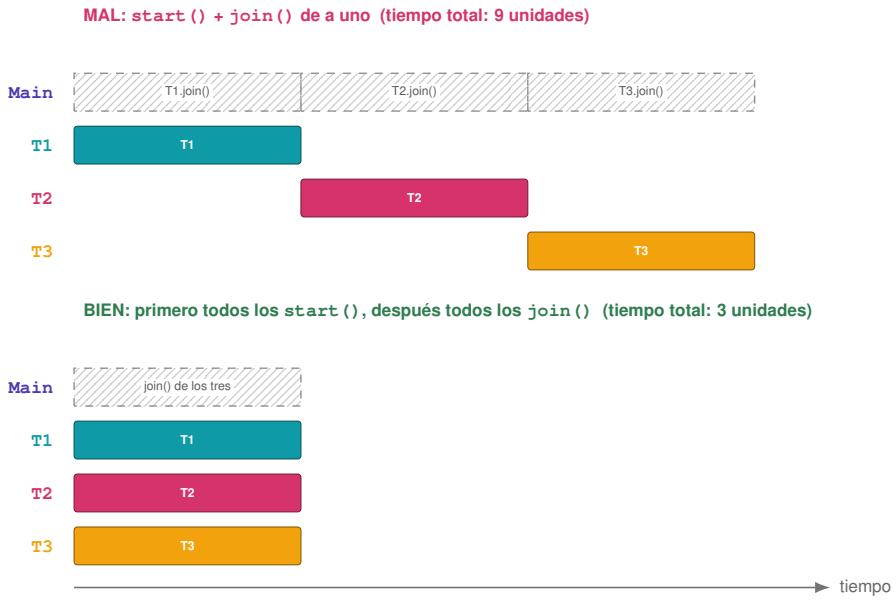
! Cuidado con dónde pones `start()` y `join()`

¿Se ejecutan estos tres *threads* en paralelo?

```
thread_1.start(); thread_1.join()
thread_2.start(); thread_2.join()
thread_3.start(); thread_3.join()
```

No. Se inicia un *thread* y se espera a que termine *antes* de iniciar el siguiente: en todo momento hay uno solo corriendo. Lo estándar es **iniciar todos primero y recién después hacer los `join()`**:

```
for t in (thread_1, thread_2, thread_3):
    t.start()                    # primero arrancan todos...
for t in (thread_1, thread_2, thread_3):
    t.join()                     # ...y despues se espera a todos
```



3.6 `is_alive()`: ¿sigue corriendo?

El método `is_alive()` indica si un *thread* **todavía está en funcionamiento**. Se usa sobre todo **después de un `join()` con `timeout`**: como `join(timeout)` no nos dice si el *thread* terminó o si simplemente se acabó el tiempo de espera, `is_alive()` es la forma de averiguarlo.

```
chisato.join(1)                  # esperamos maximo 1 segundo

if chisato.is_alive():
    print("Minions: Chisato sigue despierta. A la casa amigos!")
else:
    print("Minions: ¡Todos se durmieron! ¡A festejar!")

# Un poco despues, revisamos a cada protagonista
time.sleep(2)
for protagonista in [goku, chisato, komi, naruto, anya]:
    if protagonista.is_alive():
        print(f"Minions: {protagonista.name} aún está despiert@")
```

```
    else:
        print(f"Minions: ¡{protagonista.name} se quedó dormid@!")
```

¿Qué hace este código? Después de esperar a lo más un segundo, el programa consulta el estado real de Chisato con `is_alive()` y decide qué hacer. Más abajo, recorre a todos los protagonistas y consulta el estado de cada uno, **sin bloquearse**.

3.7 *Daemons*

Hasta ahora, el programa espera a que **todos** sus *threads* terminen antes de finalizar. Los ***daemon threads*** son la excepción.

■ Daemon thread

Un *thread* ***daemon*** es aquel que, **aunque siga corriendo, no impide que el programa termine**. El programa finaliza cuando todos los *threads* **no-*daemon*** han terminado, y en ese momento los *daemons* son **terminados inmediatamente**.

Sin los *daemons* tendríamos que llevar la cuenta de todos nuestros *threads* y asegurarnos de terminarlos uno por uno. Con ellos, podemos ponerlos a correr y **olvidarnos**: se apagarán solos cuando el programa acabe.

Se declara con el parámetro `daemon=True` en el constructor, o bien asignando el atributo `daemon` después de crear el *thread* (pero **antes** de `start()`).

```
def dormilon() -> None:
    print(f"{threading.current_thread().name} tiene sueño...")
    time.sleep(2)
    print(f"{threading.current_thread().name} se durmió.")

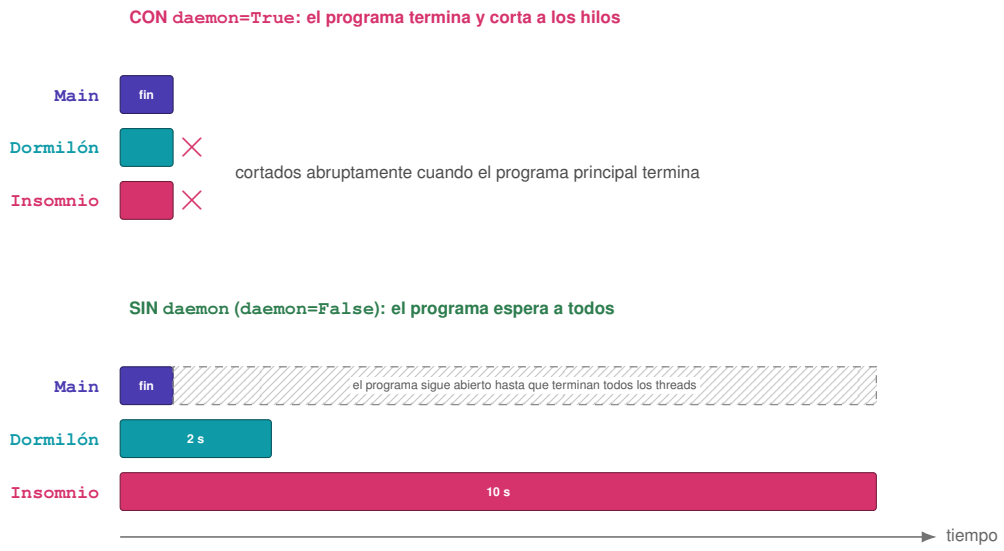
def con_insomnio() -> None:
    print(f"{threading.current_thread().name} tiene sueño...")
    time.sleep(10)
    print(f"{threading.current_thread().name} se durmió.")

# Forma 1 de hacer un thread daemon
hilo_1 = threading.Thread(name="Dormilón", target=dormilon, daemon=True)
# Forma 2 de hacer un thread daemon
hilo_2 = threading.Thread(name="Con insomnio", target=con_insomnio)
hilo_2.daemon = True

hilo_1.start()
hilo_2.start()
# el programa principal termina aqui...
```

★ Ejecútalo desde una consola

El efecto de *daemon* **no se aprecia dentro de un Jupyter Notebook** (por cómo funciona internamente). Por eso este código viene en un archivo `.py` aparte (`daemon_threads.py`) y debe ejecutarse desde una consola.



Salida con `daemon=True` y sin daemon

```
# Con daemon=True (se corta al terminar el programa principal):
Dormilón tiene sueño...
Con insomnio tiene sueño...

# Sin daemon (daemon=False): el programa no se cierra hasta el final
Dormilón tiene sueño...
Con insomnio tiene sueño...
Dormilón se durmió.              <- 2 segundos despues
Con insomnio se durmió.          <- 10 segundos despues
```

3.7.1 Esperar a un *daemon*

Es posible hacer que el programa espere a un *daemon* con `join()`, lo que tiene sentido cuando necesitamos el resultado de ese *thread* para el resto del programa.

```
hilo_1.start()
hilo_2.start()

hilo_1.join()        # espera sin limite (2 s)
hilo_2.join(5)       # espera como maximo 5 s; luego el programa termina y lo corta
```

3.7.2 No se puede cambiar el modo después de `start()`

Una vez llamado `start()`, un *thread* **no puede pasar de *daemon* a *no-daemon*** ni al revés: si se intenta, se lanza un `RuntimeError`.

```
hilo = threading.Thread(name="Dormilón", target=dormilon, daemon=True)
hilo.start()
hilo.daemon = False      # RuntimeError: cannot set daemon status of active thread
```

Si creamos una **subclase** de `Thread` que debe ser *daemon*, lo indicamos durante la inicialización:

```
class Daemon(threading.Thread):
    def __init__(self) -> None:
        super().__init__()
        self.daemon = True               # se declara al inicializar

    def run(self) -> None:
        print("Daemon thread: Empezando...")
        time.sleep(2)
        print("Daemon thread: Terminando...")

daemon = Daemon()
daemon.start()
daemon.join()
```

3.7.3 Un cuidado importante: *daemons* que imprimen

Como un *daemon* puede ser interrumpido en cualquier instante, pueden ocurrir cosas raras si estaba **a mitad de una operación con el sistema operativo**, como imprimir en consola:

```
from threading import Thread

def presentar(i):
    print("Hola soy el thread {}".format(i))

def main():
    threads = [Thread(target=presentar, daemon=True, args=(i + 1,)) for i in range(25)]
    for thread in threads:
        thread.start()

if __name__ == "__main__":
    print("Empezar programa, crearé 25 threads")
    main()
```

Aquí se crean 25 *daemons* que se presentan, y el programa principal termina justo después de lanzarlos. Es posible que, al ejecutarlo, falten mensajes (por ejemplo, el del *thread* 23) e incluso aparezca un mensaje aterrador: `Fatal Python error`.

! ¿Por qué sale un `Fatal Python error`?

Un *thread* estaba justo **en proceso de imprimir** en consola cuando el programa principal terminó y lo interrumpió. Imprimir implica enviarle una acción al sistema operativo, y cortarla a la mitad genera el error. Es **normal** que ocurra si dejamos que un *daemon* maneje `print`s y luego cerramos el programa principal de forma abrupta. Moraleja: los *daemons* sirven para trabajo "de fondo" que se puede cortar sin drama, no para tareas que deban completarse.

★ Analogía del taller

Un *daemon* es como la **radio del taller**: suena mientras haya costureras trabajando, y cuando cierran el taller se apaga en seco, sin importar si iba en medio de una canción. Nadie tiene que acordarse de apagarla, pero tampoco puede esperarse que termine lo que estaba haciendo.

3.8 *Timers*

■ Timer

La clase `Timer` es una **subclase de `Thread`** que permite **ejecutar una acción después de que pase cierto tiempo**: al iniciarse, espera el tiempo indicado y *luego* ejecuta la función. Requiere como parámetros básicos el **tiempo en segundos**, la **función** a ejecutar y sus **argumentos** (`args` o `kwargs`).

```python
from os.path import join

def tiempo_actual() -> str:
    return time.strftime("%H:%M:%S")

def mi_timer(ruta_archivo: str) -> None:
    with open(ruta_archivo) as archivo:
        for linea in archivo:
            print(f"[{tiempo_actual()}] {linea.strip()}")

t1 = threading.Timer(4.0, mi_timer, args=(join("files", "mensaje_01.txt"),))
t2 = threading.Timer(1.0, mi_timer, kwargs={"ruta_archivo": join("files", "mensaje_02.txt")})

print(f"[{tiempo_actual()}] Activando timer 1 (comienza en 4 s)")
t1.start()          # el contenido de t1 aparecerá 4 s después
print(f"[{tiempo_actual()}] Activando timer 2 (comienza en 1 s)")
t2.start()          # el contenido de t2 aparecerá 1 s después
```

¿Qué hace este código? Se crean dos *timers* que, al cumplirse su tiempo, leen e imprimen un archivo. Aunque `t1` se inicia primero, aparece **primero el contenido de `t2`**, porque su espera es más corta. Mientras tanto, el programa principal no está bloqueado.

Una ventaja de `Timer` es que, después de `start()`, podemos **cancelarlo** con `cancel()` antes de que se cumpla el tiempo:

```python
t1 = threading.Timer(4.0, mi_timer, args=("files/mensaje_01.txt",))
t1.start()
time.sleep(3)
t1.cancel()         # no alcanzo a ejecutarse: nunca veremos su contenido
```



Como un `Timer` es solo un *thread* que espera antes de partir, podemos **simular su comportamiento** con la propia clase `Thread`: guardamos el tiempo de espera y una *flag* de cancelación, y en `run()` dormimos y luego revisamos si nos cancelaron.

```python
class MiTimer(threading.Thread):
    def __init__(self, tiempo_espera: int, ruta_archivo: str) -> None:
        super().__init__()
        self.tiempo_espera = tiempo_espera
        self.ruta_archivo = ruta_archivo
        self._cancel = False

    def cancel(self) -> None:
        self._cancel = True

    def run(self) -> None:
        time.sleep(self.tiempo_espera)
        if self._cancel:                # si nos cancelaron mientras dormiamos, no hacemos nada
            return
        with open(self.ruta_archivo) as archivo:
            for linea in archivo:
                print(f"[{tiempo_actual()}] {linea.strip()}")
```

✓ Resumen de la sección: herramientas de `Thread`

- `Thread(target=f, args=..., name=..., daemon=...)`: crea el *thread*; **no** lo ejecuta.
- `start()`: lo lanza (solo una vez). `run()`: lo que ejecuta, **no se llama a mano**.
- `join(timeout)`: bloquea a *quien lo llama* hasta que el otro termine (o pase el `timeout`). `is_alive()`: ¿sigue corriendo?
- *Daemon*: no impide que el programa termine y es cortado al final. `Timer`: *thread* que espera un tiempo antes de ejecutar; se puede `cancel()`.

4 Sincronización

4.1 El problema: dos *threads*, un mismo dato

Ya sabemos crear *threads* y hacer cosas "pseudo-paralelas". ¿Qué podría salir mal? Probemos: dos *threads* que aumentan **el mismo contador** 10^6 veces cada uno. Esperaríamos que el valor final sea 2×10^6, ¿no?

```python
class Contador:
    def __init__(self) -> None:
        self.valor = 0

def sumador(contador: Contador) -> None:
    for _ in range(10 ** 6):
        contador.valor += int(1)      # el int() redundante solo hace que el error se note aqui

contador = Contador()
t1 = threading.Thread(target=sumador, args=(contador,))
t2 = threading.Thread(target=sumador, args=(contador,))

t1.start()
t2.start()
t1.join()
t2.join()

print("Listo, nuestro contador vale", contador.valor)
```

■ Una salida posible (el número exacto cambia en cada ejecución)

```
Listo, nuestro contador vale 1352874
```

¡No es 2 000 000! Como dice un viejo refrán: *"Un programador tenía un problema y decidió usar* threads. *Ahora tiene dos problemas."*

★ ¿Y si uso menos iteraciones?

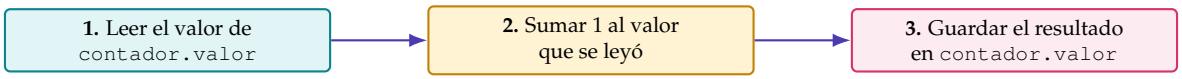
Si cambias 10^6 por 10^2, lo más probable es que el resultado salga bien, porque el primer *thread* alcanza a terminar antes de que el segundo parta. **El error no desaparece: solo se esconde.** Es lo más peligroso de estos errores: pueden pasar semanas sin manifestarse y aparecer de pronto.

4.1.1 ¿Qué pasó?

Aquí ambos *threads* intentan modificar **al mismo tiempo (concurrentemente)** un mismo valor. Esta situación se conoce como **condición de carrera** (*race condition*): el resultado depende de cómo se intercalen los *threads*. Para entenderla, recordemos que:

- Las operaciones de un *thread* **pueden ser pausadas en cualquier momento** para darle paso a otro.
- Es **imposible saber** cómo se entremezclan; lo decide el sistema operativo.

Además, `contador.valor += 1` **no es una sola operación**. A nivel de máquina, cada iteración de `sumador` hace tres pasos:



si otro *thread* se mete entre el paso 1 y el paso 3, usaremos un valor **desactualizado**

Veamos un escenario posible, paso a paso. El contador parte en 0 y cada *thread* hace una iteración (T1 hace dos en total):

#	T1	T2	contador.valor
1	lee 0		0
2	suma 1 ⇒ 1		0
3	guarda 1		1
4	lee 1		1
5	**se pausa**		1
6		lee 1	1
7		suma 1 ⇒ 2	1
8		guarda 2	2
9		lee 2	2
10		suma 1 ⇒ 3	2
11		guarda 3	3
12		**se pausa**	3
13	**se reanuda**: suma 1 ⇒ 2 (¡usa el 1 que había leído en el paso 4!)		3
14	guarda 2		2 (¡se perdió el 3!)

T1 sobreescribió con un valor **viejo** el trabajo que T2 ya había hecho. Ese es el corazón del problema: hicimos tres incrementos y el contador solo avanzó hasta 2. Repetido un millón de veces, se pierden miles de incrementos.

Para ver el intercalado en vivo, podemos separar los tres pasos con `print` y un `sleep`:

```python
def sumador(contador: Contador) -> None:
    nombre = threading.current_thread().name
    for _ in range(10):
        valor = contador.valor
        print(f"{nombre}: lee {valor}")
        nuevo_valor = valor + 1
        print(f"{nombre}: suma 1 => {nuevo_valor}")
        contador.valor = nuevo_valor
        print(f"{nombre}: guarda {nuevo_valor}")
        time.sleep(1)
```

4.1.2 Operaciones atómicas y secciones críticas

■ Atómico y sección crítica

Una operación es **atómica** si ocurre de forma **indivisible**: mientras se ejecuta, ningún otro *thread* puede intervenir. Una **sección crítica** es una secuencia de instrucciones que **no queremos que se ejecuten en paralelo**: solo **un *thread* a la vez** debe encontrarse dentro de ella.

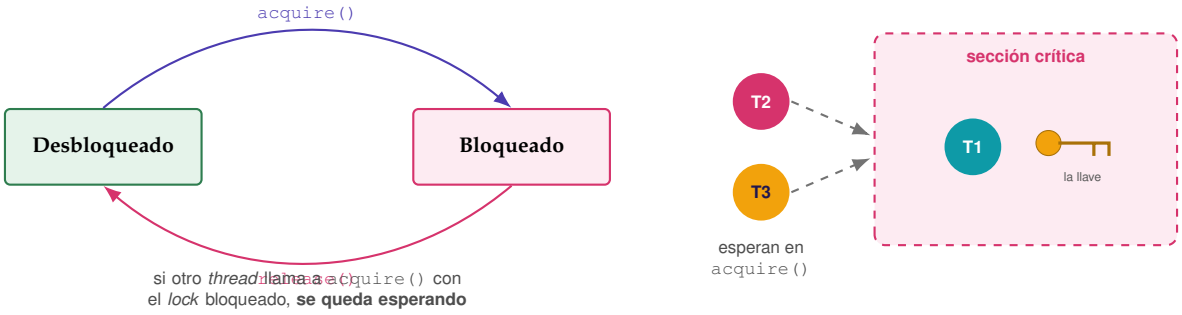
Lo que necesitamos es que `contador.valor += 1` sea una sección crítica. Este caso es solo uno de muchos en los que varios *threads* comparten recursos (archivos, variables, la consola…). Cuando hay acceso concurrente a un recurso, **solo uno** de los *threads* debe usarlo y el resto debe esperar. Para lograrlo usamos **mecanismos de sincronización** entre los *threads*.

4.2 `Lock`: una llave para la sección crítica

■ Lock

Un ***lock*** (clase `Lock` de `threading`) es una **primitiva de sincronización** que permite que solo un *thread* esté en una sección crítica a la vez. Puede estar **bloqueado** o **desbloqueado** (parte desbloqueado):

- `acquire()`: el *thread* **adquiere** el *lock* y lo deja bloqueado. Si ya estaba bloqueado por otro, el *thread* **espera** hasta que se libere.
- `release()`: el *thread* **libera** el *lock*, que queda desbloqueado y permite que otro lo adquiera.



★ Analogía del taller

Imagina que el taller tiene **una sola plancha industrial** y su llave cuelga en la puerta del cuarto de planchado. La primera costurera que llega toma la llave y entra: nadie más puede entrar (el *lock* está **bloqueado**). Las demás esperan afuera. Cuando termina, sale y **devuelve la llave** (*lock* desbloqueado), y la siguiente puede pasar.

```python
lock_global = threading.Lock()          # el MISMO lock para todos los threads

def sumador_con_seccion_critica(contador: Contador, lock: threading.Lock) -> None:
    for _ in range(10 ** 6):
        lock.acquire()                  # pedimos el lock antes de entrar
        # --- Sección crítica: aqui hay un solo thread a la vez ---
        contador.valor += int(1)
        # --- Fin de la sección crítica ---
        lock.release()                  # lo liberamos al salir

contador = Contador()
t1 = threading.Thread(target=sumador_con_seccion_critica, args=(contador, lock_global))
t2 = threading.Thread(target=sumador_con_seccion_critica, args=(contador, lock_global))
t1.start(); t2.start()
t1.join(); t2.join()
print("Listo, nuestro contador vale", contador.valor)      # 2000000
```

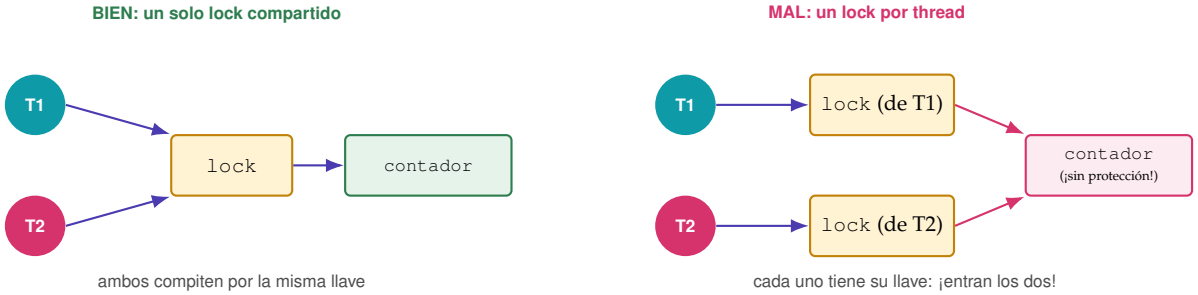
¿Qué hace este código? Cada *thread* debe adquirir el *lock* antes de tocar el contador y liberarlo al terminar. Como el *lock* es **uno solo**, mientras T1 está dentro de la sección crítica, T2 queda esperando en `acquire()` (y viceversa). Ahora el contador sí vale 2×10^6. Puede tardar unos segundos: esperar el *lock* cuesta tiempo.



4.2.1 Atención: el *lock* debe ser compartido

El *lock* que usan los *threads* **debe ser el mismo**. Si cada *thread* crea su propio *lock*, todos pasan el `acquire()` sin esperar a nadie y entran a la sección crítica a la vez: no nos protegemos de nada.

```python
def sumador_con_seccion_critica_mala(contador: Contador) -> None:
    lock = threading.Lock()          # ¡MAL! cada thread crea SU PROPIO lock
    for _ in range(10 ** 6):
        lock.acquire()
        contador.valor += int(1)     # ambos threads siguen entrando a la vez
        lock.release()
```



4.2.2 *Locks* y *context managers*: `with lock:`

Los *locks* también funcionan como ***context managers*** con la sentencia `with`. Es el mismo `with` el que llama a `acquire()` al entrar y a `release()` al salir, de forma implícita:

```python
lock_global = threading.Lock()

def sumador(contador: Contador, lock: threading.Lock) -> None:
    nombre = threading.current_thread().name
    for _ in range(10):
        with lock:                                  # acquire() al entrar
            # --- Sección crítica ---
            valor = contador.valor
            print(f"{nombre}: lee {valor}")
            nuevo_valor = valor + 1
            print(f"{nombre}: suma 1 => {nuevo_valor}")
            contador.valor = nuevo_valor
            print(f"{nombre}: guarda {nuevo_valor}")
            time.sleep(1)
        # release() al salir del with

contador = Contador()
t1 = threading.Thread(name="T1", target=sumador, args=(contador, lock_global))
t2 = threading.Thread(name="T2", target=sumador, args=(contador, lock_global))
t1.start(); t2.start()
t1.join(); t2.join()
```

¿Qué hace este código? Es el mismo ejemplo con `print`s de antes, pero ahora los tres pasos (leer, sumar, guardar) ocurren **dentro** del `with lock`. En la salida verás que cada *thread* completa sus tres pasos seguidos, sin que el otro se meta en medio.

★ Prefiere `with` sobre `acquire()` / `release()`

Si te olvidas de llamar a `release()` (o si el código de la sección crítica lanza una excepción antes de llegar a él), el *lock* queda **bloqueado para siempre** y todos los demás *threads* se quedan esperando. Con `with`, el *lock* se libera automáticamente al salir del bloque, incluso si hubo una excepción. Además, procura que la sección crítica sea **lo más corta posible**: mientras un *thread* está dentro, todos los demás esperan.

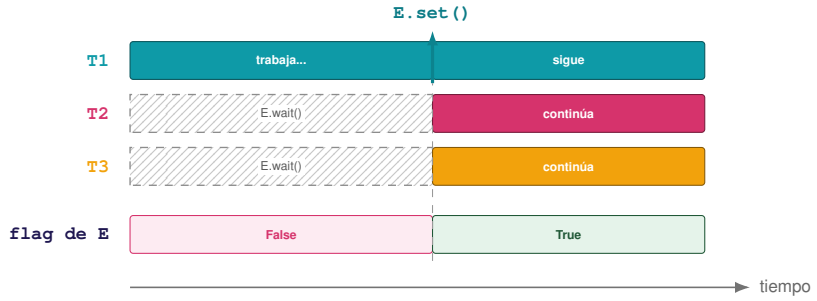
4.3 Señales entre *threads*: `Event`

Otro patrón muy común es la **comunicación de eventos**: que un *thread* espere a que ocurra algo para continuar. Con `join()` podemos esperar el **término** de otro *thread*, pero `join()` no resuelve:

- ¿Y si queremos esperar un evento que ocurre **durante** la vida de otro *thread*?
- ¿Y si el segundo *thread* **no tiene que terminar** para que el primero continúe?
- ¿Cómo hacemos que un *thread* **espere a que otro le avise**?

Para eso existen los objetos `Event`: uno de los mecanismos más simples de comunicación entre *threads*. **Un *thread* hace una señal y otros esperan a que ocurra.** Un `Event` tiene un ***flag*** interno: `True` si la señal está activa y `False` si no.

Método	Acción	¿Qué hace?
`wait()`	**Esperar**	El *thread* **queda en pausa** hasta que otro active la señal. Si la señal **ya estaba activa**, sigue inmediatamente sin esperar.
`set()`	**Avisar**	Deja el *flag* en `True`: activa la señal (y despierta a quienes esperaban).
`clear()`	**Resetear**	Deja el *flag* en `False`: apaga la señal, para poder reutilizarla.
`is_set()`	**Revisar**	Retorna `True` o `False` según el *flag*, **sin pausar** el *thread* (a diferencia de `wait()`).



En el diagrama, T2 y T3 usan `E.wait()` para esperar el evento `E`. Cuando T1 llama a `E.set()`, **ambos se despiertan a la vez** y continúan. T1 nunca tuvo que terminar.

★ Analogía del taller

Es como la **campana del taller**: las costureras que ya terminaron su parte esperan junto a la mesa hasta que la encargada toca la campana (`set()`) para avisar "la tela está lista". Todas las que estaban esperando parten al mismo tiempo, y quien llegue después de la campana ni siquiera necesita esperar.

4.3.1 Ejemplo: sincronizar audio y video

Supón que un *thread* carga el audio (3 s) y otro carga el video (10 s), y queremos que **ambos empiecen a reproducir a la vez**. Cada uno avisa cuando está listo y luego espera al otro:

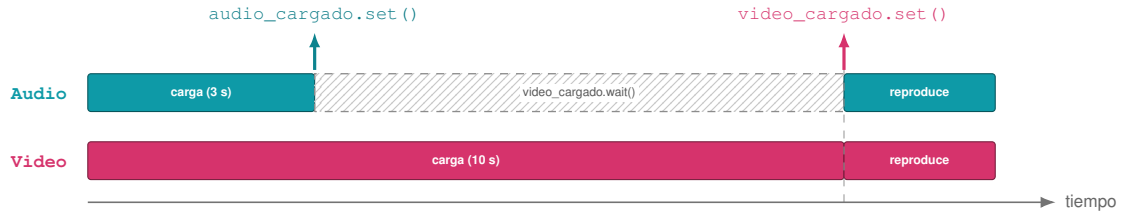
```
video_cargado = threading.Event()       # "el video ya está listo"
audio_cargado = threading.Event()       # "el audio ya está listo"

def reproducir_video(nombre: str) -> None:
    print("Cargando video. Se demorará 10 segundos")
    time.sleep(10)
    print("Video cargado. Esperando audio")
    video_cargado.set()                 # AVISA que el video está listo
    audio_cargado.wait()                # ESPERA al audio
    print("Comenzando reproducción del video")

def reproducir_audio(nombre: str) -> None:
```

```
    print("Cargando audio. Se demorará 3 segundos")
    time.sleep(3)
    print("Audio cargado. Esperando video")
    audio_cargado.set()                # AVISA que el audio está listo
    video_cargado.wait()               # ESPERA al video
    print("Comenzando reproducción del audio")

t1 = threading.Thread(target=reproducir_audio, args=("Zen Zen Zense",))
t2 = threading.Thread(target=reproducir_video, args=("Zen Zen Zense",))
t1.start()
t2.start()
t1.join()
t2.join()
```



¿Qué hace este código? El audio termina de cargar a los 3 s, avisa con `audio_cargado.set()` y se queda esperando el video. El video termina a los 10 s, avisa con `video_cargado.set()` y, como el audio **ya había avisado antes**, su `audio_cargado.wait()` pasa **de inmediato**. Ambos comienzan **casi simultáneamente**. Sin los eventos, el audio habría partido 7 segundos antes.

4.3.2 Ejemplo: trabajador y jefe

Este ejemplo usa `set()`, `clear()`, `wait()` e `is_set()` en un mismo problema. Hay dos *threads*: un **trabajador** y un **jefe**. El trabajador trabaja una cantidad de semanas y **cada 4 semanas pide su sueldo**, pausando su trabajo hasta recibirlo. El jefe sigue trabajando mientras el trabajador no termine, y **cada semana revisa** si le pidieron el sueldo; si es así, paga y espera la confirmación antes de seguir. Cuando el trabajador termina, avisa y ambos finalizan.

```
solicitar_pago = threading.Event()         # el trabajador pidió su pago mensual
entregar_pago = threading.Event()          # el jefe pagó
confirmar_recepcion = threading.Event()    # el trabajador recibió el pago
finalizar_trabajo = threading.Event()      # el trabajador terminó todas sus labores

def trabajar_trabajador(semanas: int) -> None:
    for semana in range(1, semanas + 1):
        print(f"[Trabajador] Trabajando la semana #{semana}")
        time.sleep(1)
        if not semana % 4:                          # fin de mes
            print("[Trabajador] ¡Jefe! Ya estamos a fin de mes")
            confirmar_recepcion.clear()
            solicitar_pago.set()                    # pide el pago
            entregar_pago.wait()                    # espera el pago
            print("[Trabajador] A seguir trabajando pues")
            confirmar_recepcion.set()               # confirma que recibió
            solicitar_pago.clear()                  # apaga su solicitud
    print("[Trabajador] Terminé todas mis labores.")
    finalizar_trabajo.set()

def trabajar_jefe() -> None:
    finalizado = False
    while not finalizado:
        if finalizar_trabajo.is_set():              # revisa SIN detenerse
            finalizado = True
        else:
            print("[Jefe] Trabajando...")
            if solicitar_pago.is_set():             # ¿me pidieron el sueldo?
                print("[Jefe] Aquí tiene su dinero estimado")
                entregar_pago.set()
                confirmar_recepcion.wait()          # espera la confirmación
                entregar_pago.clear()
            else:
                print("[Jefe] Todavía no me piden el pago de este mes")
            time.sleep(1)
```

```
    print("[Jefe] Podemos dar el proyecto por finalizado")

t_jefe = threading.Thread(target=trabajar_jefe)
t_trabajador = threading.Thread(target=trabajar_trabajador, args=(12,))
t_jefe.start()
t_trabajador.start()
t_jefe.join()
t_trabajador.join()
```



El **apretón de manos** (*handshake*) se repite cada 4 semanas hasta que el trabajador avisa `finalizar_trabajo.set()`

★ ¿Por qué el jefe usa `is_set()` y no `wait()`?

El jefe debe seguir trabajando cada semana **aunque nadie le haya pedido nada**. Si usara `solicitar_pago.wait()`, se quedaría pausado hasta que llegara la solicitud y dejaría de trabajar. Con `is_set()` solo *consulta* el *flag* y sigue. **Regla:** `wait()` para *esperar* de verdad; `is_set()` para *mirar* sin detenerse.

4.4 Otros mecanismos de coordinación

Python ofrece otras formas de coordinar *threads*, adecuadas para otras situaciones. **No las veremos en detalle en el curso**, pero conviene saber que existen (están en la documentación de `threading`):

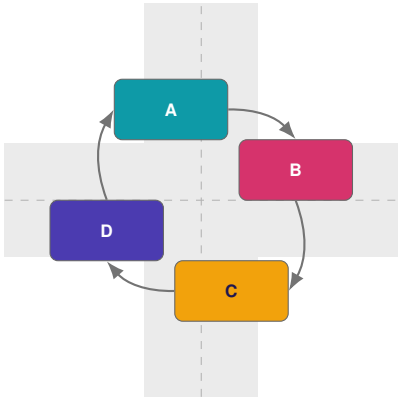
Objeto	Idea general
`RLock`	*Lock* **reentrante**: el mismo *thread* puede adquirirlo varias veces sin bloquearse a sí mismo.
`Condition`	Permite esperar hasta que se cumpla una **condición** y avisar cuando cambia.
`Semaphore`	Como un *lock*, pero deja pasar hasta N ***threads*** a la vez.
`Barrier`	Punto de encuentro: los *threads* esperan hasta que llegan **todos** los que se necesitan.

5 *Deadlocks*

Hemos visto formas de coordinar *threads* o de hacer que uno **espere** a otro. Pero, por error, pueden darse situaciones en que dos o más *threads* **se esperan mutuamente** y ninguno avanza jamás. Eso se llama ***deadlock*** o **interbloqueo**.

■ Deadlock (interbloqueo)

Situación en que dos o más *threads* quedan **bloqueados para siempre** porque cada uno espera algo que solo puede entregar otro de los que también está esperando. Hay una **espera circular**: nadie puede dar el primer paso.



A espera que **B** avance, **B** espera a **C**, **C** espera a **D** y **D** espera a **A**. Ninguno puede moverse: un interbloqueo de tráfico es exactamente lo mismo que uno entre *threads*.

★ Analogía del taller

Una costurera tiene las **tijeras** y necesita el **hilo**; otra tiene el **hilo** y necesita las **tijeras**. Cada una espera que la otra suelte lo suyo primero. Pueden quedarse así toda la tarde.

5.1 Ejemplo 1: *deadlock* con *locks*

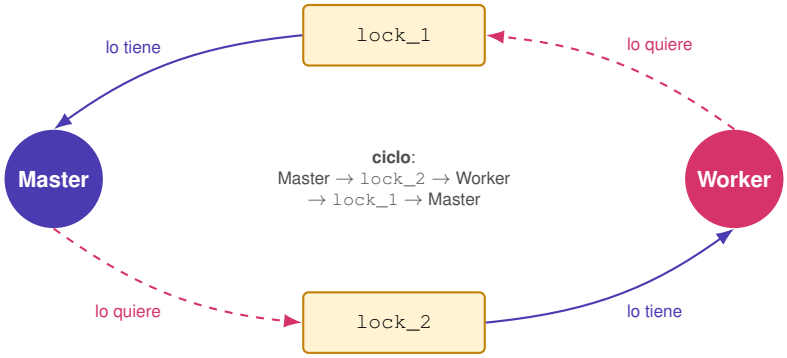
```python
lock_1 = threading.Lock()
lock_2 = threading.Lock()

def master() -> None:
    time.sleep(2)
    print("Master: adquiriendo lock_1")
    with lock_1:
        time.sleep(2)
        print("Master: adquiriendo lock_2")
        with lock_2:
            print("Master: ¡trabajando!")

def worker() -> None:
    time.sleep(1.5)
    print("Worker: adquiriendo lock_2")
    with lock_2:
        time.sleep(2)
        print("Worker: adquiriendo lock_1")
        with lock_1:
            print("Worker: ¡trabajando!")

t1 = threading.Thread(target=master)
t2 = threading.Thread(target=worker)
t1.start()
t2.start()
```

¿Qué pasa? *Master* alcanza a adquirir `lock_1` y *worker* alcanza a adquirir `lock_2`. Luego *master* intenta adquirir `lock_2` y debe esperar a que *worker* lo libere; pero *worker* no liberará `lock_2` sin antes adquirir `lock_1`. **Ninguno de los dos llega a decir que está trabajando.**



Tiempo	Master	Worker
1,5 s	(durmiendo)	adquiere `lock_2`
2 s	adquiere `lock_1`	(durmiendo dentro de `lock_2`)
3,5 s	(durmiendo dentro de `lock_1`)	quiere `lock_1` ⇒ **queda bloqueado**
4 s	quiere `lock_2` ⇒ **queda bloqueado**	(bloqueado)

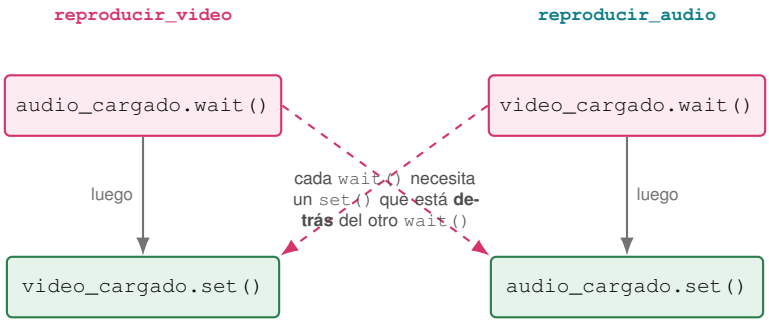
5.2 Ejemplo 2: *deadlock* con *events*

Es el mismo ejemplo de audio y video de la Sección 4.3, pero **cambiando el orden**: ahora cada *thread* espera al otro *antes* de avisar que hizo lo suyo.

```python
def reproducir_video(nombre: str) -> None:
    time.sleep(10)
    print("Video cargado. Esperando audio")
    audio_cargado.wait()        # ESPERA al audio...
    video_cargado.set()         # ...y RECIEN DESPUES avisa (tarde)

def reproducir_audio(nombre: str) -> None:
    time.sleep(3)
    print("Audio cargado. Esperando video")
    video_cargado.wait()        # ESPERA al video...
    audio_cargado.set()         # ...y RECIEN DESPUES avisa (tarde)

# Si hicieras t1.join() / t2.join() aqui, el programa nunca terminaria.
```



¿Qué pasa? `reproducir_video` espera que `audio_cargado` se active antes de activar `video_cargado`. Pero `reproducir_audio` espera que `video_cargado` se active antes de activar `audio_cargado`. Ninguno llega nunca a su `set()`.

! Cómo evitar los deadlocks

Lo esencial es **saber que ocurren** y programar con cuidado. Algunas prácticas habituales (consejos de ayudante, además del material del curso):

- Si necesitas **varios *locks***, adquiérelos **siempre en el mismo orden** en todos los *threads* (en el ejemplo 1, ambos deberían pedir primero `lock_1` y luego `lock_2`).
- Con *events*, **avisa primero y espera después** (como en el ejemplo bueno de audio y video).
- Evita quedarte esperando algo mientras retienes un recurso que otro necesita, y mantén las secciones críticas cortas.

✓ Resumen de la sección: sincronización

- Varios *threads* modificando un mismo dato sin coordinarse producen una **condición de carrera**: el resultado depende del intercalado y puede ser incorrecto.
- Una **sección crítica** debe ejecutarla un solo *thread* a la vez. Se protege con un `Lock` **compartido** (idealmente con `with lock:`).
- Un `Event` sirve para **avisar** y **esperar señales** (`set`, `wait`, `clear`, `is_set`) sin que el otro *thread* tenga que terminar.
- Si dos *threads* se esperan mutuamente, hay un ***deadlock***: revisa el orden de tus `acquire`/`wait`.

6 Ejemplos y aplicaciones

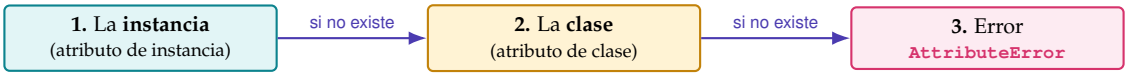
6.1 Un *lock* como atributo de una subclase de `Thread`

Una forma ordenada de tener **un mismo *lock* para todos los *threads*** de una clase es definirlo como **atributo de clase**. Así todos los objetos lo comparten sin tener que pasarlo como argumento.

6.1.1 Paréntesis: atributos de clase y atributos de instancia

- Un **atributo de instancia** se define con `self.atributo = valor` (normalmente en `__init__`). Cada objeto tiene el suyo.
- Un **atributo de clase** se define **al nivel de los métodos**, dentro de la clase. Es **uno solo**, compartido por todas las instancias, y se accede con `Clase.atributo` (o con `self.atributo` desde una instancia).

  Cuando escribimos `self.atributo`, Python busca el valor en este orden:



6.1.2 Ejemplo: escribir en un mismo archivo desde varios *threads*

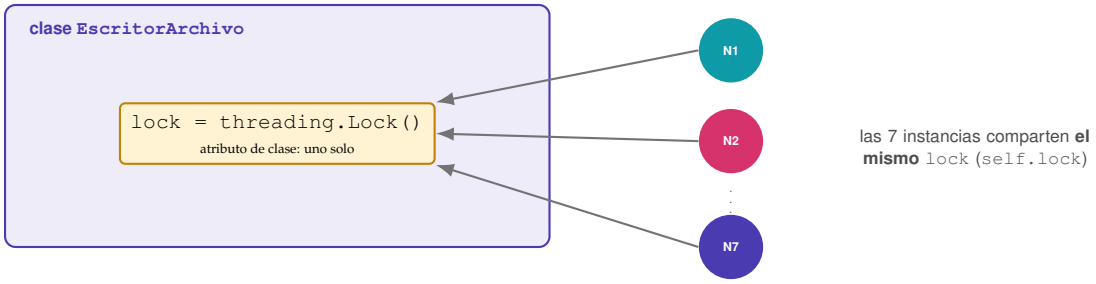
```python
import threading, time, os, typing
from random import random

class EscritorArchivo(threading.Thread):
    # Atributo de CLASE: un solo lock, compartido por todas las instancias.
    # Accesible como EscritorArchivo.lock o self.lock
    lock = threading.Lock()

    def __init__(self, numero: int, archivo: typing.TextIO) -> None:
        super().__init__(name=f"EscritorArchivo número {numero}")
        self.numero = numero
        self.archivo = archivo

    def run(self) -> None:
        print(f"[{self.name}] ¡Comenzó a trabajar!")
        for _ in range(self.numero):
            with self.lock:                          # acceso al lock compartido
                self.archivo.write(f"Línea escrita por # {self.name}\n")
                print(f"[{self.name}] ¡escribió una línea!")
            time.sleep(random())                     # espera aleatoria en [0, 1)

with open(os.path.join("files", "salida.txt"), "w", encoding="utf-8") as archivo:
    threads = [EscritorArchivo(i, archivo) for i in range(1, 8)]
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()          # esperamos a todos ANTES de cerrar el archivo
    print("Todos los threads terminaron. Revise el resultado en files/salida.txt")
```



¿Qué hace este código? Se crean 7 *threads*; el *thread* número k escribe k líneas en el mismo archivo. Como todos comparten `EscritorArchivo.lock`, cada escritura ocurre dentro de una sección crítica y **nunca se mezclan dos líneas**. El archivo final tendrá $1 + 2 + \cdots + 7 = 28$ líneas, en un orden que cambia en cada ejecución.

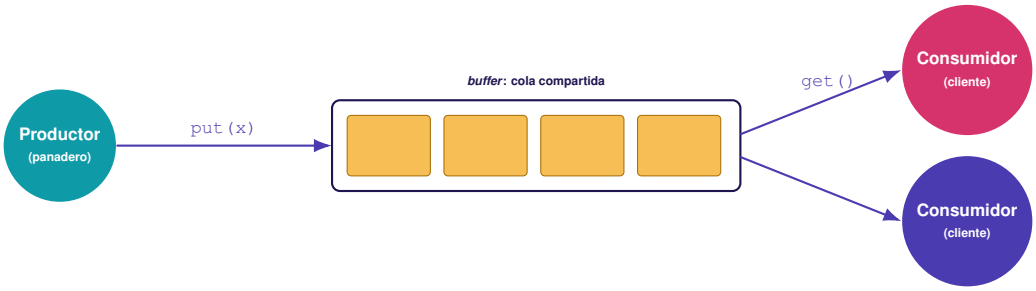
★ ¿Por qué el `join()` antes de cerrar el archivo?

El `with open(...)` cierra el archivo al terminar su bloque. Si el programa principal llegara ahí mientras los *threads* aún escriben, intentarían escribir en un archivo **ya cerrado**. Por eso se espera a todos con `join()` dentro del `with`.

6.2 Patrón productor-consumidor

■ Productor-consumidor

Patrón común de programación concurrente en que dos o más *threads* acceden a un mismo espacio de almacenamiento o ***buffer***. Los **productores ponen** ítems en el *buffer* y los **consumidores** los **sacan**. Es una forma de **comunicar** *threads*. Lo habitual es implementar el *buffer* como una **cola sincronizada** (o cola *thread-safe*).



6.2.1 Primera versión: `deque` y un *lock*

Un `deque` permite agregar y sacar elementos de forma segura entre *threads*, pero **nada nos asegura** que, si vimos que había un objeto para sacar, ese objeto *todavía esté* cuando vayamos a sacarlo (otro cliente pudo llevárselo en medio). Por eso debemos asegurar **nosotros** con un *lock* que "revisar si hay algo" y "sacarlo" sea una operación **atómica**.

Ejemplo: el productor es un **panadero** y los consumidores son **clientes** de un supermercado.

```
from collections import deque
from random import choices

piezas_de_pan = deque()
lock_sacar_pan = threading.Lock()

def panadero() -> None:
    for partida in range(3):                # hace pan 3 veces
        time.sleep(5)                       # se demora 5 s por partida
        piezas = choices(["Marraqueta", "Baguette", "Hallulla"], k=5)
        print("[Panadero] Produje 5 piezas de pan en la partida", partida)
        piezas_de_pan.extend(piezas)

def cliente(i: int) -> None:
    print(f"[Cliente {i}] ¡Quiero pan!")
    while True:
        with lock_sacar_pan:                # "ver si hay" + "sacar" deben ser atómicos
            if piezas_de_pan:
```

```
                mi_pan = piezas_de_pan.popleft()
                print(f"[Cliente {i}] ¡Saqué mi {mi_pan}!")
                break                       # ya tiene su pan: termina

thread_panadero = threading.Thread(target=panadero)
threads_clientes = [threading.Thread(target=cliente, args=(i,)) for i in range(15)]

thread_panadero.start()
for t in threads_clientes:
    t.start()
for t in threads_clientes:
    t.join()
```

¿Qué hace este código? El panadero produce 5 piezas cada 5 segundos, tres veces. Cada uno de los 15 clientes revisa una y otra vez si hay pan; cuando hay, lo saca dentro del `with lock_sacar_pan` para que nadie se lo quite entre el "ver" y el "sacar".

! Funciona, pero es ineficiente: espera activa

Mientras no hay pan, cada cliente **vuelve a revisar de inmediato**, consumiendo tiempo de CPU en vano (**espera activa**, *busy waiting*). Sería mucho mejor que el panadero *avisara* cuando hay pan listo y los clientes durmieran mientras tanto (aun así, tendrían que verificar que quede pan, porque puede haber más clientes que piezas).

6.2.2 Segunda versión: `queue.Queue`

Python trae un módulo `queue` con una cola **hecha para varios *threads***, que resuelve justo este problema. Sus métodos principales:

Método	¿Qué hace?
`put(x)`	Agrega un ítem al final de la cola (*push*).
`get()`	Remueve y retorna un ítem de la cola (*pop*). Lo interesante: **espera** (bloqueando al *thread*) hasta que haya algo que sacar.
`task_done()`	Se llama **una vez por cada ítem** extraído que ya fue procesado.
`join()`	El *thread* que lo llama queda en pausa hasta que **todos** los ítems de la cola hayan sido procesados.

```
from queue import Queue

piezas_de_pan = Queue()

def panadero() -> None:
    for partida in range(3):
        time.sleep(5)
        print("[Panadero] Produje 5 piezas de pan en la partida", partida)
        for pieza in choices(["Marraqueta", "Baguette", "Hallulla"], k=5):
            piezas_de_pan.put(pieza)

def cliente(i: int) -> None:
    print(f"[Cliente {i}] ¡Quiero pan!")
    mi_pan = piezas_de_pan.get()            # se duerme hasta que haya pan
    print(f"[Cliente {i}] ¡Saqué mi {mi_pan}!")
    piezas_de_pan.task_done()               # ya procesé este ítem
```

¿Qué hace este código? El panadero solo hace `put`. Cada cliente hace un único `get()`: si no hay pan, el *thread* **se duerme** sin gastar CPU y despierta cuando el panadero agrega una pieza. Ya no necesitamos *lock* propio ni ciclo de revisión: la `Queue` maneja la sincronización internamente.

	deque + Lock	Queue
Sincronización	La escribimos nosotros.	Ya viene incluida.
Cuando la cola está vacía	El cliente revisa una y otra vez (**espera activa**).	`get()` **duerme** al *thread* hasta que haya algo.
Riesgo de errores	Alto: olvidar el *lock*, mala atomicidad.	Bajo.

! Si no hay suficiente pan, alguien espera para siempre

En el ejemplo, 3 partidas × 5 piezas = 15 piezas para 15 clientes: justo alcanza. Si hubiera 16 clientes, uno quedaría bloqueado en `get()` **para siempre** (y, como no es *daemon*, el programa nunca terminaría). Con `Queue` hay que asegurarse de que a los consumidores siempre les llegue algo (o de tener una forma de decirles que ya no habrá más).

★ Analogía del taller

La `Queue` es la **cinta transportadora** entre la costurera (productora) y la planchadora (consumidora): la costurera deja piezas terminadas en la cinta, y la planchadora toma la siguiente cuando puede. Si la cinta está vacía, la planchadora simplemente espera (`get()`) sin dar vueltas por el taller preguntando "¿ya hay algo?".

6.2.3 Una cola por *thread*: enviar avisos a todos

En el ejemplo del pan, solo **un** *thread* recibía cada ítem. Pero a veces queremos que **todos** reciban lo mismo, como un aviso del curso. En ese caso, en vez de una `Queue` compartida, usamos **una `Queue` por usuario**, y cada *thread* está atento solo a la suya.

```
import queue, random

class Estudiante(threading.Thread):
    def __init__(self, nombre: str) -> None:
        threading.Thread.__init__(self, name=nombre)
        self.cola_correos = queue.Queue()          # su propia cola de correo

    def notificar(self, aviso: str) -> None:
        self.cola_correos.put(aviso)               # otros threads le dejan avisos

    def run(self) -> None:
        while True:
            aviso = self.cola_correos.get()        # espera a que llegue un aviso
            print(f"[{self.name}] Ohhh tengo un correo")
            time.sleep(random.randint(0, 2))
            print(f"[{self.name}] Su contenido es: {aviso}")
            self.cola_correos.task_done()

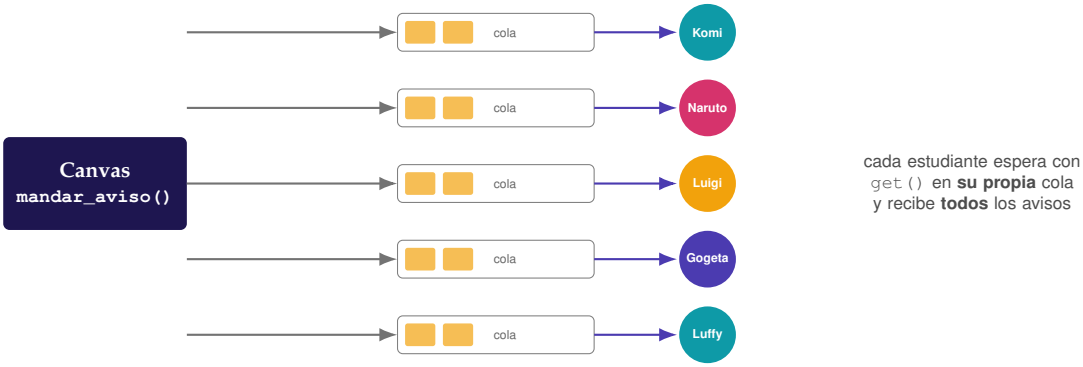
class Canvas:
    def __init__(self) -> None:
        self.estudiantes = []

    def agregar_estudiante(self, estudiante: Estudiante) -> None:
        self.estudiantes.append(estudiante)

    def mandar_aviso(self, aviso: str) -> None:
        print("[CANVAS] Enviando un aviso a los estudiantes")
        for estudiante in self.estudiantes:
            estudiante.notificar(aviso)

canvas = Canvas()
for nombre in ["Komi", "Naruto", "Luigi", "Gogeta", "Luffy"]:
    estudiante = Estudiante(nombre)
    canvas.agregar_estudiante(estudiante)
    estudiante.start()

canvas.mandar_aviso("Aviso Semanal")
canvas.mandar_aviso("Recuerden hacer la evaluación de Threading!")
```



★ Ojo con los `while True`

Los `Estudiante` de este ejemplo no son *daemon* y su `run()` es un ciclo infinito. En un **script**, el programa **nunca terminaría**, porque siempre queda algún *thread* no-*daemon* vivo (en el *notebook* no se nota). En la práctica, un *thread* así se marca como *daemon* o se le da una forma de detenerse (por ejemplo, con un `Event`). Además, con `Queue` puedes comunicar *threads* no solo con objetos cualquiera, sino con **mensajes con un formato** que todos entiendan.

6.3 Simulaciones

Otra aplicación habitual de los *threads* es **simular entidades con comportamiento simultáneo**. Junto con *locks* y señales, permite modelar situaciones muy variadas.

Ejemplo: un **estudiante** que programa línea por línea y, con cierta probabilidad, tiene una **duda**; y un **profesor** que espera y **solo responde cuando hay una duda**.

```python
from threading import Event, Thread
from time import sleep
from random import randint

class Estudiante(Thread):
    def __init__(self, lineas_estudio: int, senal_duda: Event, senal_resuelto: Event):
        super().__init__()
        self.max_lineas = lineas_estudio          # lineas de codigo que programara
        self.senal_duda = senal_duda              # "tengo una duda"
        self.senal_resuelto = senal_resuelto      # "el profe resolvió mi duda"

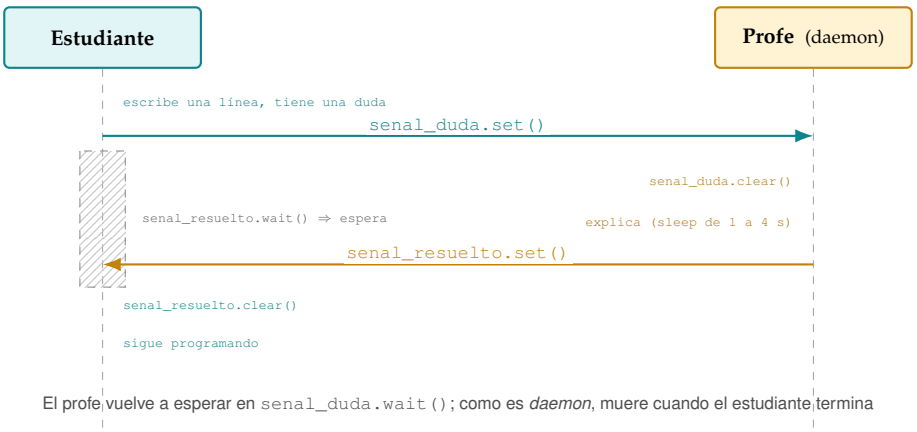
    def run(self) -> None:
        print("[Estudiante] ¡A programar!")
        lineas = 0
        while lineas < self.max_lineas:
            sleep(1)
            lineas += 1
            print("[Estudiante] --------- Una linea escrita")
            # Despues de cada línea puede tener una duda; mientras más líneas
            # lleva, menos probable es.
            if randint(1, self.max_lineas) >= lineas:
                print("[Estudiante] Tengo una duda profe :(")
                self.senal_duda.set()             # avisa que tiene una duda
                self.senal_resuelto.wait()        # espera la respuesta
                self.senal_resuelto.clear()       # reinicia la señal para reusarla
                print("[Estudiante] Ahora entiendo profe, gracias :)")
        print("[Estudiante] ¡Terminé de programar!")

class Profe(Thread):
    def __init__(self, senal_duda: Event, senal_resuelto: Event) -> None:
        super().__init__()
        self.senal_duda = senal_duda
        self.senal_resuelto = senal_resuelto
        self.daemon = True                        # el profe debe ser daemon

    def run(self) -> None:
        while True:                               # espera por siempre...
            self.senal_duda.wait()                # ...a que haya una duda
            self.senal_duda.clear()               # reinicia la señal
            print("[Profesor] Mira, lo que pasa es que...")
            tiempo = randint(1, 4)
            sleep(tiempo)                         # explica y se demora un poco
```

```
            print(f"[Profesor]... {tiempo} segundos después")
            self.senal_resuelto.set()              # avisa que resolvió la duda

senal_tengo_una_duda = Event()
senal_duda_resuelta = Event()
estudiante = Estudiante(10, senal_tengo_una_duda, senal_duda_resuelta)
profe = Profe(senal_tengo_una_duda, senal_duda_resuelta)
profe.start()
estudiante.start()
```



¿Qué hace este código? Hay dos señales: `senal_duda` (del estudiante al profe) y `senal_resuelto` (del profe al estudiante). Cada vez que el estudiante tiene una duda, la activa y se **pausa** esperando la respuesta; el profe, que dormía en `wait()`, se despierta, **apaga** la señal para poder reutilizarla, explica, y activa la señal de vuelta. Fíjate en dos decisiones de diseño:

- **Cada *Event* se limpia con `clear()`** por quien lo consumió, para poder volver a usarlo en la siguiente duda.
- **El profe es *daemon***: su `run()` es un ciclo infinito, así que si no fuera *daemon* el programa jamás terminaría. Al ser *daemon*, se apaga solo cuando el estudiante (el único no-*daemon*) termina.

✓ Resumen de la sección: aplicaciones

- Un *lock* **de clase** (`lock = threading.Lock()` al nivel de la clase) lo comparten todas las instancias de una subclase de `Thread`.
- **Productor-consumidor**: productores hacen `put`, consumidores hacen `get`. Con `deque` tú sincronizas; con `Queue` la sincronización ya viene incluida y `get()` espera sin gastar CPU.
- Una `Queue` **por *thread*** sirve para enviar el **mismo** mensaje a todos.
- Los *threads* + `Event` permiten **simular** entidades que se coordinan entre sí; los *daemons* son ideales para las que "viven para siempre".

7 Hoja de referencia

Herramienta	Qué hace	Detalle importante
`Thread(target=f)`	Crea un *thread* que ejecutará `f`.	No parte solo. Argumentos: `args`, `kwargs`.
`start()`	Lanza el *thread*.	Solo una vez (si no, `RuntimeError`).
`run()`	Instrucciones del *thread* (en subclases).	Se sobreescribe; **no** se llama a mano.
`join(timeout)`	Bloquea a *quien lo llama* hasta que el otro termine.	Con `timeout` espera como máximo ese tiempo.
`is_alive()`	¿El *thread* sigue corriendo?	Útil tras un `join(timeout)`.
`daemon=True`	No impide que el programa termine.	Se fija antes de `start()`; se corta al final.
`current_thread()`	Referencia al *thread* que ejecuta ese código.	`.name` da su nombre.
`Timer(t, f, args)`	Ejecuta `f` tras `t` segundos.	Se puede `cancel()` antes de que se cumpla.
`Lock()`	Exclusión mutua para la sección crítica.	`with lock:`. Debe ser **el mismo** para todos.
`Event()`	Señal entre *threads*.	`set`, `clear`, `wait` (bloquea), `is_set` (no bloquea).
`Queue()`	Cola *thread-safe*.	`put`, `get` (espera), `task_done`, `join`.

¿Qué herramienta uso?

Algo debe correr a la par del programa	`Thread` + `start()`
Esperar que otro *thread* termine	`join()` (tras iniciar **todos**)
Varios *threads* tocan el mismo recurso	`Lock` compartido
Esperar un aviso sin que el otro termine	`Event`
Pasar datos entre *threads*	`Queue`
Ejecutar algo más tarde	`Timer`
Trabajo de fondo que puede cortarse	*Daemon*

! Errores frecuentes

- Olvidar `start()`, o llamar `run()` en vez de `start()`.
- Llamar `start()` dos veces, o cambiar `daemon` después de `start()` (`RuntimeError`).
- Poner `start()` y `join()` de a uno: queda secuencial.
- Olvidar `super().__init__()` en una subclase de `Thread`.
- Que cada *thread* cree **su propio** `Lock`: no protege nada.
- Olvidar `release()` (usa `with`) o dejar secciones críticas largas.
- Asumir un orden de ejecución entre *threads*.
- *Deadlocks*: locks en distinto orden, o `wait()` antes de `set()`.
- `get()` sobre una `Queue` que nunca recibirá nada.

✓ Checklist de código con *threads*

1. ¿Hice `start()` en todos y *después* `join()` en todos?
2. ¿Todo recurso compartido está protegido por **un mismo** `Lock`?
3. ¿Uso `with lock:` con secciones críticas cortas?
4. ¿Cada `wait()` tiene un `set()` que realmente puede ocurrir?
5. ¿Los *threads* con `while True` son *daemon* o pueden detenerse?

Fin del resumen • ¡Que ningún hilo se te enrede!