

# Tema 2 - Hilos (Threads)

---

## 2.1.- Introducción y fundamentos.

- Un proceso se descompone en otros subprocesos.
- **Relación general entre núcleos e hilos**
  - **1 núcleo = 1 hilo** → en procesadores **sin multihilo**.
  - **1 núcleo = 2 hilos** → en procesadores **con multihilo simultáneo (SMT o Hyper-Threading, como los Intel o AMD modernos)**.

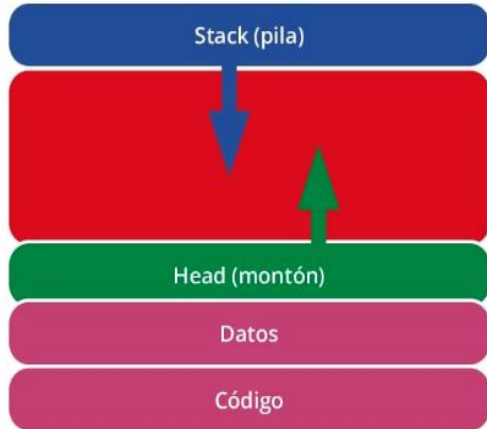
Procesador	Núcleos	Hilos por núcleo	Total de hilos
Intel Core i5-10400	6	2	12
AMD Ryzen 5 5600	6	2	12
Intel Core i3-10100	4	2	8
Apple M1	8	1 (no usa SMT)	8

## 2.1.- Introducción y fundamentos.

- Un **hilo** es una **unidad lógica de ejecución**, o sea, una tarea que el sistema operativo puede ejecutar en paralelo.
- El programador es el encargado de, sobre todo:
  - Particionar programa a ejecutar en paralelo.
  - Usar el recurso apropiado para gestionar los recursos compartidos o críticos (exclusión mutua).
  - Recursos adecuados para sincronizar los hilos.
- Todo proceso tiene un hilo principal (main thread) del cual se pueden crear otros secundarios.

## 2.1.- Introducción y fundamentos.

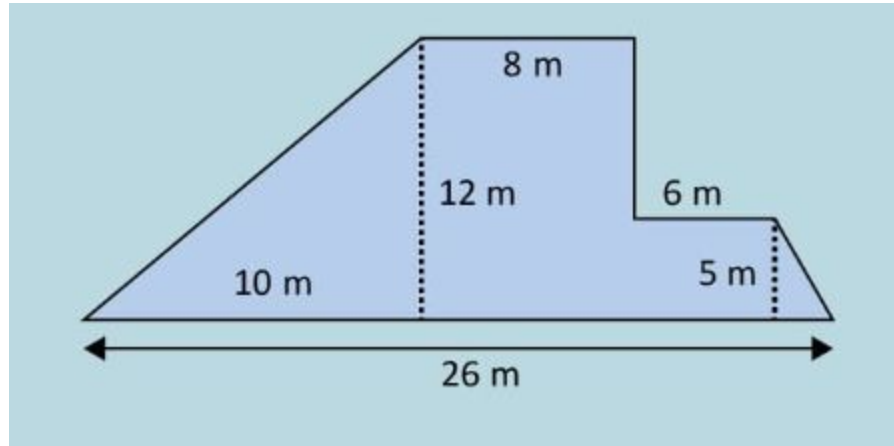
- Un hilo comparte con su proceso todos los recursos indicados en el PCB. En la memoria común están:



```
+-----+
|  Stack (pila)  | ← variables locales, llamadas a funciones
+-----+
|    Heap       | ← memoria dinámica (malloc, new, etc.)
+-----+
| Data segment (.bss) | ← variables globales y estáticas
+-----+
|  Text segment (.text) | ← código del programa
+-----+
```

## 2.1.- Introducción y fundamentos.

- Ejercicio: Plantear solución usando hilos para optimizar:



## 2.2.- Creación y puesta en ejecución.

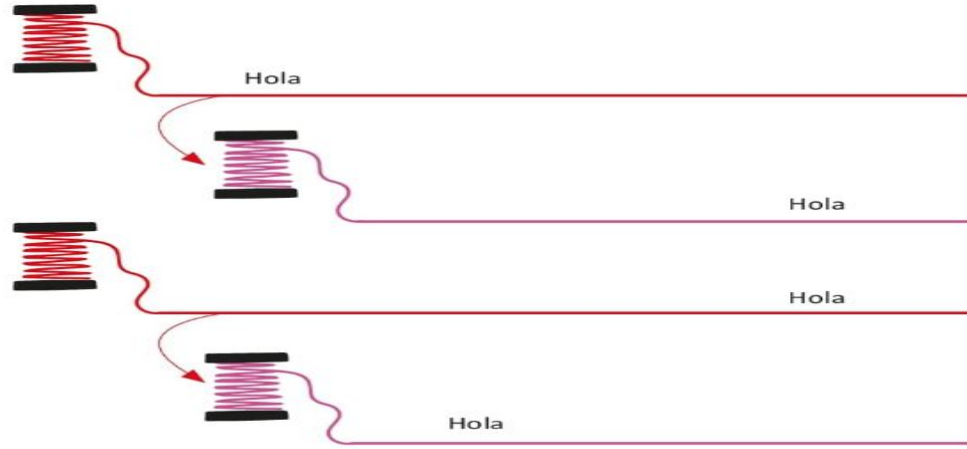
```
import threading
import time
import random

#método al que se va a asociar el hilo
def Saludo():
    time.sleep(random.random())
    print ('Hola en el hijo')

t = threading.Thread(target=Saludo)
t.start()
time.sleep(random.random())
print ("Hola en el padre") #impresión en el hilo principal
```

**Ejecutar y probarlo.**

## 2.2.- Creación y puesta en ejecución.



**Ejercicio:** Intenta hacer el ejercicio primero.

## 2.2.- Creación y puesta en ejecución.

'''

**En este código se aprende cómo almacenar hilos en listas (arrays). Nos será útil. Escribir y probar.**

'''

```
import threading
```

```
def actividad():  
    print ("Escribo desde un hilo")
```

```
print ("INICIO")  
hilos = list() #En java sería 'ArrayList<¿tipo?> lista = new ArrayList<>()
```

```
for i in range(50):  
    t = threading.Thread(target=actividad)  
    hilos.append(t) #En java sería hilos.add(t)  
    t.start()
```

```
print ("ESCRIBO EN PRINCIPAL")
```

## 2.3.- No determinismo.

'''

**Aquí se puede observar no determinismo -> No se da la misma salida en ejecuciones diferentes.**

'''

```
import threading
```

```
import time
```

```
def escribeY():
```

```
    for _ in range(1000):
```

```
        time.sleep(3) #Para provocar cambios de contexto. Duerme el hilo, sacándolo del procesador.
```

```
        print("Y")
```

```
hilo = threading.Thread(target=escribeY)
```

```
hilo.start()
```

```
for _ in range(1000):
```

```
    time.sleep(3)
```

```
    print("X")
```

## 2.3.- Equivalencia en Java.

```
public class HilosEjemplo {  
    public static void main(String[] args) {  
  
        // Creamos un hilo que imprime "Y"  
        Thread hiloY = new Thread() -> {  
            for (int i = 0; i < 1000; i++) {  
                Thread.sleep(3000); // Duerme 3 segundos  
                System.out.println("Y");  
            }  
        });  
  
        // Iniciamos el hilo  
        hiloY.start();  
  
        // El hilo principal imprime "X"  
        for (int i = 0; i < 1000; i++) {  
            Thread.sleep(3000); // Duerme 3 segundos  
            System.out.println("X");  
        }  
    }  
}
```

## 2.3.- Variables compartidas.

- A las variables compartidas se debe acceder en **exclusión mutua**

```
import threading

contador = 0

def incrementar():
    global contador
    for _ in range(1000000):
        contador += 1 # ¡No es atómico!

hilos = []
for i in range(10):
    hilo = threading.Thread(target=incrementar)
    hilos.append(hilo)
    hilo.start()

for hilo in hilos:
    hilo.join()

print(f"Contador esperado: 1000000, obtenido: {contador}")
```

## 2.3.- Variables compartidas.

- El problema: `contador += 1` no es atómico
  - La operación `contador += 1` parece simple, pero internamente se ejecuta en tres pasos:
    1. Leer el valor actual de `contador`
    2. Incrementar el valor leído
    3. Escribir el nuevo valor de vuelta a `contador`
- Qué puede salir mal

```
# Ejemplo de condición de carrera:  
# Hilo A lee contador = 100  
# Hilo B lee contador = 100  
# Hilo A escribe 101  
# Hilo B escribe 101
```

```
# ¡Se perdieron incrementos!
```

## 2.3.- Paso de parámetros.

- Dos formas. Una usando **args** (que es una lista de parámetros) o bien por diccionario usando **kwargs** (qué es pares de clave-valor). Son equivalentes y más clara **args**.

```
import threading

def imprimeNombre(nombre):
    print( nombre + 'Rodríguez')

nombres = ["Pepe", "Amancio", "Pantoja", "Nicolasio"]
hilos = []
for nombre in nombres:
    hilo = threading.Thread(target=imprimeNombre, args=(nombre,))
    hilo.start()
    hilos.append(hilo)

#Esperar a que todos los hilos terminen
for hilo in hilos:
    hilo.join()
print ("Todos los hilos han terminado")
```

## 2.3.- Paso de parámetros.

- **Ejercicio:** Crea un hilo que le pasemos dos números enteros  $n1$  y  $n2$  y escriba la secuencia comprendida entre ambos números, si  $n1 < n2$ . Desde el hilo principal debe mostrar un mensaje con el resultado de la resta.

## 2.4.- Propiedades.

- **threading.current\_thread().name** -> Nos dice el nombre del hilo. Es de lectura/escritura.
- **threading.current\_thread().ident** -> solo lectura.
- Podemos cambiarle el nombre a un hilo (el sistema le da uno por defecto):
  - `eseHilo = threading.Thread(target=funcion_hilo, name="miHilo")`
- Y también podemos cambiarlo de esta manera:
  - `eseHilo.name` = "eseHiloeee"
- **enumerate()** -> accedemos a los hilos que actualmente están corriendo bajo el proceso actual.
- **activate\_count()** -> Nos dice el número de hilos activos.

## 2.4.- Propiedades.

- **Ejercicio:** Crea un proceso que realice lo siguiente :
  - Crea 5 hilos y los echa a andar.
  - Cada hilo muestra su nombre y su identificador y después hace un `time.sleep(3)`.
  - Al crear cada hilo deberás cambiar el nombre del proceso por **"eseHilooo-identificador"**, donde **identificador** lo sacarás de la propiedad **ident**.
  - Usa **activate\_count()** para decirnos el número de procesos que hay.
  - Usa **enumerate** para listar los procesos mostrando su nombre y su identificador.
  - Espera a que todos sus hilos acaben y entonces muestra el mensaje **"Finalizado y a otra cosa mariposa"**.

## 2.4.- Propiedades - DAEMON.

- Un hilo daemon finaliza inmediatamente cuando finaliza el proceso que lo lanza.
- Por defecto los hilos tienen la propiedad daemon a false, eso significa que es el padre el que debe esperar, o no, a que finalice el hijo.
- Se usan hilos **DAEMON** para:
  - Tareas de fondo no críticas.
  - Monitoreo.
  - Logs.
  - Actualizaciones periódicas.
  - Conexiones de red temporales.
- Se usan hilo **NO DAEMON** para:
  - Procesamiento de datos importante.
  - Guardado de archivos críticos.
  - Cálculos que deben completarse.
  - Tareas donde los resultados son esenciales.

## 2.4.- Propiedades - DAEMON.

- **Probar en modo Daemon y no Daemon.**

```
import threading
import time

def hilo():
    for i in range(10):
        print("Hilo no daemon (Foreground)")
        time.sleep(1)

t = threading.Thread(target=hilo)
t.daemon = True
t.start()

print("Hilo principal")
```

## 2.4.- Propiedades - DAEMON.

```
import threading
import time
```

```
def guardado_automtico():
```

```
    contador = 0
```

```
    while True:
```

```
        time.sleep(10) # Guardar cada 10 segundos
```

```
        contador += 1
```

```
        print(f" Guardado automático #{contador}")
```

```
# Daemon: No importa si se interrumpe el guardado
```

```
guardador = threading.Thread(target=guardado_automtico, daemon=True)
```

```
guardador.start()
```

```
# Simular edición de documento
```

```
print(f" Editando documento...")
```

```
for i in range(3):
```

```
    time.sleep(3)
```

```
    print(f" Escribiendo línea {i+1}...")
```

```
print("Documento cerrado - el guardado automático se detiene")
```

## 2.5.- Herencia clase Thread.

- Crear una clase para el hilo.
- Equivalente a lo que hemos hecho hasta ahora, pero mucho más potente y legible.
- El constructor **`__init__()`** se invoca en el momento de la instanciación. Aquí pasamos los parámetros al método.
- El método **`run()`**, que será invocado, bien por una llamada directa, o bien a través del método **`start()`**
- En una clase en python, todos los métodos llevan delante **`self`**

## 2.5.- Herencia clase Thread.

```
import threading

#-----aquí comienza la clase.
class miHilo(threading.Thread):
    #Aquí inicializamos variables de clase, atributos, con valores de fuera.
    def __init__(self, num):
        super(miHilo, self).__init__()
        self.numero = num
        self.nombre = ""

    #Aquí va el código del hilo.
    def run(self):
        if self.nombre == "hiloPar":
            print(f"par: {self.numero}")
        else:
            print(f"impar:{self.numero}")

#-----aquí termina la clase y comienza el programa para usar la clase.

for i in range(4):
    hilo = miHilo(i)
    if (i % 2 == 0):
        hilo.nombre = "hiloPar"
    else:
        hilo.nombre = "hiloImpar"
    hilo.start()
```

## 2.5.- Herencia clase Thread.

- **Ejercicio:** Crea un proceso que realice lo siguiente, usando la clase Thread :
  - Crea 5 hilos y los echa a andar.
  - Cada hilo muestra su nombre y su identificador y después hace un `time.sleep(3)`.
  - En un bucle desde el programa principal muestras el nombre e identificador de cada hilo, usando una propiedades de la clase.
  - A continuación, dentro del bucle, cambias el nombre del hilo por "esePedazoHiloX", donde X es el identificador del hilo.
  - Vuelves a recorrer cada uno de los hilos y muestras el nombre.
  - Esperas a que todos los hilos terminen y en el proceso principal muestras el mensaje **"Finalizado todo usando la clase thread"**

## 2.6.- Sincronismo. Join.

- El proceso que lo contiene se queda parado hasta que el hilo referenciado no termine.

```
import threading
import time

def miHilo(palabra, contador):
    time.sleep(3)
    for _ in range(contador):
        print(palabra)

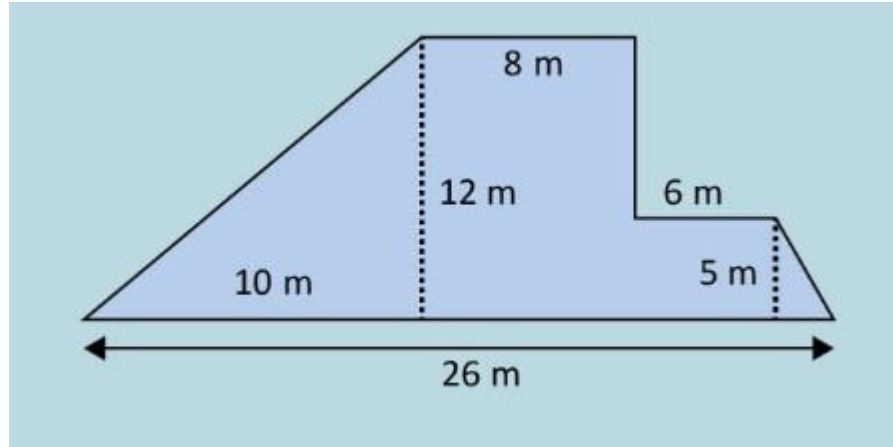
hilo1 = threading.Thread(target=miHilo, args=("hola", 20,))
hilo2 = threading.Thread(target=miHilo, args=("adiós", 15,))

hilo1.start()
hilo2.start()

#hasta que no finalicen los dos hilos, no sale este mensaje.
print("Hilos acabados")
```

## 2.6.1- Sincronismo. Join.

- Hacer el programa anterior con y sin join.
- Hacer el ejercicio propuesto al principio de tema. Los hilos derivaran de la clase `threading.Thread`



## 2.6.2- Sincronismo. Lock.

- **Región crítica:** conjunto de líneas de código en las que se accede a un recurso compartido, como puede ser una variable, a la que solo un hilo o proceso lo haga al mismo tiempo.
- Una instrucción en un lenguaje de alto nivel puede dar lugar a una o **varias instrucciones en código intermedio (bytecodes)**. A su vez, **cada bytecode** puede generar **múltiples sentencias en código máquina**.
- En la región crítica debe haber el menor número de líneas de código para que la concurrencia sea lo máxima posible. Solo el código estrictamente necesario.

## 2.6.2- Sincronismo. Lock.

```
import threading #sin usar lock.
import time

def suma_uno():
    global g
    a=g
    time.sleep(0.001)
    g = a+1

def suma_tres():
    global g
    a=g
    time.sleep(0.001)
    g =a+3

hilos = []
for func in [suma_uno,suma_tres]:
    hilos.append(Thread(target=func))
    hilos[-1].start() # -1 señala al último elemento de la lista.

for hilo in hilos:
    hilo.join()

print(g)
```

## 2.6.2- Sincronismo. Lock.

- Métodos de **threading.Lock()**:
  - **lock.acquire()** -> coge el recurso y nadie más puede entrar.
  - **lock.release()** -> libera el recurso.
  - **lock.locked()** -> True o false según el lock actual esté en uso o no.

## 2.6.2- Sincronismo. Lock.

```
import threading #usando lock.
import time

def suma_uno():
    global g
    lock.acquire()
    a=g
    time.sleep(0.001)
    g = a+1
    lock.release()

def suma_tres():
    global g
    lock.acquire()
    a=g
    time.sleep(0.001)
    g =a+3
    lock.release()

lock = threading.Lock()
hilos = []
for func in [suma_uno,suma_tres]:
    hilos.append(Thread(target=func))
    hilos[-1].start() # -1 señala al último elemento de la lista.

for hilo in hilos:
    hilo.join()

print(g)
```

## 2.6.2- Sincronismo. Lock.

```
import threading #Otra forma
import time

def suma_uno():
    global g
    with lock:
        a=g
        time.sleep(0.001)
        g = a+1

def suma_tres():
    global g
    with lock:
        a=g
        time.sleep(0.001)
        g =a+3

lock = threading.Lock()
hilos = []
for func in [suma_uno,suma_tres]:
    hilos.append(Thread(target=func))
    hilos[-1].start() # -1 señala al último elemento de la lista.

for hilo in hilos:
    hilo.join()

print(g)
```

## 2.6.2- Sincronismo. Lock.

- **Ejercicio 1:** Mete el programa anterior en una clase `threading.Thread`
- **Ejercicio 2:** Contador compartido. Incrementar un contador global desde varios hilos sin que se pierdan actualizaciones.
- **Ejercicio 3:** Acceso a un archivo compartido. Varios hilos escriben en el mismo archivo de texto. Usamos Lock para que no se corrompa la escritura.
  - Se simulará un fichero usando una variable de texto, que se llamará 'fichero', a la que se le añade "El hiloX escribe la línea T".
  - Cada hilo escribe un número aleatorio de líneas consecutivas, no más de 5 líneas.
  - Crea 5 hilos.
  - Usa un `sleep`, poner 1 segundo, para forzar un cambio de contexto cuando el hilo esté escribiendo en la variable 'fichero'.
  - Se crean los hilos de forma aleatoria y se les hace `join` a todos para que el programa principal pueda mostrar el contenido de la variable de texto 'fichero'.
  - Prueba después a ejecutarlo sin el lock ¿Qué ocurre? ¿Qué pasa si aumentas el `sleep` hasta 10?
- **Ejercicio 4:** Simulación de retiro bancario. Simular múltiples hilos modificando el saldo de una cuenta bancaria compartida. Las cuentas bancarias son clases y se usan una clase para crear los hilos que modifican las cuentas.

## 2.6.2- Sincronismo. Rlock.

- Se usa muy poco, así que no lo damos.

## 2.6.2- Sincronismo. Condicionales.

- Comunicación entre hilos para bloquear/desbloquear recursos.
- Los hilos se coordinan mediante los **condition**
- Permite que un hilo espere hasta que otro hilo le notifique que algo ha cambiado (por ejemplo, que hay datos disponibles, que un recurso está libre, etc.).
- Un objeto Condition combina:
  - Un **bloqueo (lock)** para proteger el acceso a recursos compartidos.
  - Una **cola de espera** para los hilos que están esperando una señal.
- Cuando se usa:
  - Varios hilos necesitan coordinarse.
  - ***Unos hilos producen datos y otros consumen o esperan datos.***
  - Ejemplos: productor-consumidor, cola de tareas, etc....

## 2.6.2- Sincronismo. Condicionales.

```
import threading
import time
import random.
```

```
def productor(condition, datos):
```

```
    with condition:
```

```
        print("Productor: Creando datos para el consumidor")
```

```
        tiempo = random.randint(1,5)
```

```
        time.sleep(tiempo) #Nos ayuda a simular que está trabajando.
```

```
        datos = f"Trabajo realizado en {tiempo} segundos"
```

```
        condition.notify() # Despierta a un solo hilo. Para despertar a varios usar NOTIFY_ALL()
```

```
        print("Productor: Acabado. comunicado al consumidor.")
```

```
def consumidor(condition, datos):
```

```
    with condition:
```

```
        print("Consumidor: esperando datos...")
```

```
        condition.wait() # Espera hasta que alguien llame a notify()
```

```
        print(f"Consumidor: datos recibidos -> {datos}")
```

```
#crear estructuras compartidos.
```

```
condition = threading.Condition()
```

```
datos = None
```

```
# Crear hilos
```

```
elConsumidor = threading.Thread(target=consumidor, args=(condition, datos))
```

```
elProductor = threading.Thread(target=productor, args=(condition,datos))
```

```
#Activarlos
```

```
elConsumidor.start()
```

```
elProductor.start()
```

## 2.6.2- Sincronismo. Condicionales.

- **Ejercicio 1:** Simula una cola de impresión.
  - Habrá un hilo que imprimirá, es decir meterá un mensaje en una lista. Usa `time.sleep` para simular diferentes tiempos de escritura.
  - La impresora será el hilo que imprime, realiza un `print` de ese mensaje. Sacando el mensaje de la lista. La impresora está constantemente esperando a que haya un trabajo para imprimir. Usa `time.sleep` para simular diferentes tiempos de impresión.
- **Ejercicio 2:** Semáforo de cruce peatonal. Imagina un sistema simple donde:
  - Un coche pasa por una intersección si no hay peatones cruzando.
  - Un peatón quiere cruzar, pero solo lo hace si no hay coches cruzando.
  - No pueden cruzar ambos al mismo tiempo por seguridad.
  - Reglas:
    - Si un peatón está cruzando, los coches deben esperar.
    - Si un coche está cruzando, los peatones deben esperar.
    - Solo puede cruzar uno a la vez (ya sea peatón o coche).
    - Se puede tener múltiples hilos representando peatones o coches que quieren cruzar en cualquier momento.

## 2.6.2- Sincronismo. Condicionales.

- **Ejercicio 3:** Situación más real, en el semáforo pueden pasar varios coches a la vez y peatones también, pero no pueden pasar a la vez peatones y coches:
  - Un coche pasa por una intersección si no hay peatones cruzando.
  - Un peatón quiere cruzar, pero solo lo hace si no hay coches cruzando.
  - No pueden cruzar ambos al mismo tiempo por seguridad.
  - Reglas:
    - Si, al menos, un peatón está cruzando, los coches deben esperar.
    - Si, al menos, un coche está cruzando, los peatones deben esperar.
    - Se puede tener múltiples hilos representando peatones o coches que quieren cruzar en cualquier momento.

## 2.6.2- Sincronismo. Condicionales.

- ¿Por qué se usa while y no if al esperar una condición?
  - En sincronización de hilos, “esperar” no siempre significa que la condición se cumplió.
  - Código correcto:

```
with condicion:  
    while not lista:  
        condicion.wait()  
    dato = lista.pop()
```
  - Código incorrecto:

```
with condicion:  
    if not lista:  
        condicion.wait()  
    dato = lista.pop()
```
- **Razones:**
  - Despertares espurios: un hilo puede despertarse sin notify().
  - Varios consumidores: otro hilo puede consumir antes el recurso.
  - Buena práctica universal: siempre revalidar la condición con while.
- **En resumen:**
  - “while asegura que la condición sea verdadera de verdad antes de continuar.”

## 2.6.2- Sincronismo. Semáforos.

- Un **semáforo** es un contador que **limita el número de hilos** que pueden acceder simultáneamente a un recurso compartido.
- Si el contador llega a 0, los demás hilos deben **esperar** hasta que otro hilo libere el semáforo.
- Un semáforo funciona como un “guardia” que deja pasar solo a cierto número de hilos al mismo tiempo.

```
import threading
import time

# Creamos un semáforo que permite 2 hilos simultáneamente
semaforo = threading.Semaphore(2)

def tarea(nombre):
    print(f"{nombre} esperando acceso...")
    with semaforo: # Entra si hay lugar
        print(f"{nombre} accede al recurso")
        time.sleep(2)
    print(f"{nombre} libera el recurso")

# Creamos varios hilos
for i in range(4):
    threading.Thread(target=tarea, args=(f"Hilo-{i+1}",)).start()
```

## 2.6.2- Sincronismo. Semáforos.

- Ejemplos de uso en la vida real
  - **Control de acceso a recursos limitados**
    - **Ejemplo:** Varios hilos quieren usar una impresora, pero solo hay **2 impresoras disponibles**.
    - **Solución:** Un semáforo con valor 2 permite que solo dos hilos impriman a la vez; los demás esperan hasta que una impresora quede libre.
  - **Conexiones simultáneas a un servidor**
    - **Ejemplo:** Un servidor web solo puede atender **100 usuarios** a la vez.
    - **Solución:** Un semáforo limita el número de hilos de atención concurrentes, evitando que el servidor colapse por exceso de peticiones.
  - **Acceso controlado a una base de datos**
    - **Ejemplo:** Muchos hilos intentan escribir datos al mismo tiempo.
    - **Solución:** El semáforo asegura que solo un número limitado de hilos escriba simultáneamente, previniendo inconsistencias o bloqueos.
  - **Sistemas de tráfico o producción**
    - **Ejemplo:** En una línea de ensamblaje o cruce de tráfico, solo cierto número de vehículos o productos puede pasar por una etapa al mismo tiempo.
    - **Solución:** El semáforo actúa literalmente como un **control de paso**, sincronizando el flujo.
  - **Gestión de recursos en la nube**
    - **Ejemplo:** Varias máquinas virtuales comparten un grupo limitado de CPUs o GPUs.
    - **Solución:** Los semáforos limitan cuántas instancias acceden al hardware simultáneamente.

## 2.6.2- Sincronismo. Semáforos.

- **Entrada limitada a una sala**
  - Solo 2 personas pueden entrar al mismo tiempo.

```
import threading, time

sala = threading.Semaphore(2)

def persona(nombre):
    print(f"{nombre} quiere entrar...")
    with sala:
        print(f"{nombre} entra a la sala")
        time.sleep(2)
        print(f"{nombre} sale de la sala")

for i in range(5):
    threading.Thread(target=persona, args=(f"Persona-{i+1}",)).start()
```

## 2.6.2- Sincronismo. Semáforos.

- **Impresora compartida**

- Solo una impresora disponible; los hilos imprimen uno por uno.

```
import threading, time

impresora = threading.Semaphore(1)

def imprimir(nombre):
    with impresora:
        print(f'{nombre} está imprimiendo...')
        time.sleep(1)
        print(f'{nombre} terminó de imprimir.')

for i in range(3):
    threading.Thread(target=imprimir, args=(f'Hilo-{i+1}',)).start()
```

## 2.6.2- Sincronismo. Semáforos.

- **Ejercicio 1: Conexiones a un servidor web**
  - Un servidor puede atender como máximo 3 conexiones simultáneas (para no saturarse).
  - Varios clientes intentan conectarse.
  - Si ya hay 3 conectados, los demás deben esperar.
- **Ejercicio 2: Acceso a base de datos**
  - Una base de datos permite que hasta 2 consultas se ejecuten a la vez.
  - Cada consulta tarda unos segundos en completarse.
  - Si hay más de 2 peticiones, deben esperar turno.

## 2.6.3- Sincronismo. Eventos.

- Un objeto Event actúa como una bandera compartida entre hilos.
- Un hilo puede esperar hasta que otro señala que algo ha ocurrido.
- Es una forma de sincronización unidireccional (uno o varios hilos esperan a que otro dé la señal).
- Se comporta como un interruptor binario:
  - Estado inicial a false.
  - **set()** -> se activa evento (bandera true)-> hilos bloqueados se desbloquean.
  - **clear()** -> se desactiva evento ( bandera false) ->hilos se bloquean.
  - **wait(timeout=None)** -> bloquea el hilo hasta que el evento se active o expire el timeout.
  - **is\_set()** -> Devuelve True si el evento está activado.

### 2.6.3- Sincronismo. Eventos.

- **Ejercicio 1:** Un hilo prepara datos y otros hilos esperan a que los datos estén listos para procesarlos.

## 2.6.3- Sincronismo. Eventos.

```
evento = threading.Event()
```

```
class productor(threading.Thread):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
    def run(self):
```

```
        print("Soy el productor. Estoy produciendo datos....'To quisqui' esperando")
```

```
        time.sleep(random.uniform(0.5,1))
```

```
        print("El productor da paso a los consumidores")
```

```
        evento.set() #Desbloqueo a los hilos que estuviesen esperando.
```

```
class consumidor(threading.Thread):
```

```
    def __init__(self, nombre):
```

```
        super().__init__()
```

```
        self.nombre = nombre
```

```
    def run(self):
```

```
        print(f"Soy el consumidor {self.nombre} y espero a que el productor termine")
```

```
        evento.wait()
```

```
        print("Ya puedo entrar a consumir")
```

```
#Creo los consumidores.
```

```
consumidores=[]
```

```
for c in range(5):
```

```
    cons = consumidor(c+1)
```

```
    cons.start()
```

```
    consumidores.append(cons)
```

```
#Lanzo el productor.
```

```
pr=productor()
```

```
pr.start()
```

```
#El padre espera a que todos terminen.
```

```
for c in consumidores:
```

```
    cons.join()
```

```
pr.join()
```

```
print("Programa terminado")
```

## 2.6.3- Sincronismo. Eventos.

- Son útiles para:
  - Coordinar varios hilos dentro de un servicio (por ejemplo, que uno no empiece hasta que otro haya inicializado recursos).
  - Sincronizar tareas de E/S (esperar a que llegue una señal de otro proceso o cliente).
  - Implementar controladores de estado (esperar a que se cumpla una condición antes de continuar).
  - Diseñar sistemas de productores/consumidores donde los productores avisan a los consumidores con un Event.

## 2.6.3- Sincronismo. Eventos.

- **Ejercicio 2:** Un hilo trabaja continuamente hasta que llega otro proceso y lo detiene. Hay que usar `is_set()`.
- **Ejercicio 3:** Un hilo controla un cruce. 10 segundos es para que pasen hilos peatones solo, 5 segundos es para que pasen hilos coches solamente. Llegan continuamente tanto coches como peatones.
- **Ejercicio 4:** Hacer que un hilo aborte la ejecución de otro. El hilo a abortar está funcionando en un bucle infinito y un segundo hilo lo para cuando lo necesita.

## 2.6.3- Sincronismo. Eventos.

<b>Mecanismo</b>	<b>Cuándo usarlo</b>
<b>Lock</b>	Para proteger acceso a un recurso compartido (evita condiciones de carrera).
<b>Semaphore</b>	Para limitar número de hilos concurrentes.
<b>Event</b>	Para sincronizar el inicio/fin de tareas, o esperar señales de otros hilos.
<b>Condition</b>	Para sincronización más compleja (varias condiciones, notificaciones específicas).