GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

Unidad 1.4

Programación multihilo

Índice

Sumario

Licencia

1. Paralelismo y variables compartidas

Cuando trabajas con hilos en Python, es posible que tengas múltiples hilos ejecutándose en paralelo. Uno de los desafíos en la programación concurrente es compartir datos entre estos hilos de manera segura, ya que el acceso concurrente a recursos compartidos puede llevar a condiciones de carrera y resultados inesperados.

Recursos que comparten los hilos:

- Variables globales
- Archivos abiertos (la entrada y salida estándar también son archivos abiertos)
- Otros recursos

Dentro de cada hilo se pueden declarar variables locales que ve sólo ese hilo, el resto de hilos **no tienen acceso a es variable local**. Por eso es importante distinguir bien entre variables globales y variables locales. Temas a tener en cuenta:

- Dónde inicializar una variable global.
- Diferencias entre acceder a una variable global sólo para leer su valor y acceder a una variable global para modificar su valor.

1.1. Variables locales y variables globales

A continuación tenéis unas cuantas pautas sobre la declaración de variables globales y locales en el contexto de hilos en Python (threads):

1. **Variables Locales:**
    - Las variables locales son aquellas que se definen dentro de una función y no están disponibles fuera de esa función.
    - Cada hilo tiene su propio conjunto de variables locales. Los otros hilos no pueden ni acceder ni ver las variables locales de un hilo en concreto.
    - No es necesario preocuparse por el conflicto de variables locales entre hilos, ya que cada hilo tiene su propio espacio de nombres de función.
2. **Variables Globales:**
    - Las variables globales son compartidas por todos los hilos en un programa.
    - Si un hilo necesita leer el valor de una variable global, no hay problemas y no se requiere ninguna declaración especial.
    - Sin embargo, si un hilo intenta modificar el valor de una variable global, se debe usar la palabra clave **`global`** para indicar que se está modificando la variable global y no creando una variable local con el mismo nombre.
    - El acceso de diferentes hilos a una misma variables globales puede generar resultados inesperados y es por ello que se propondrán diversos modos de control.

Para ver dónde y cómo se declaran las variables globales, vamos a trabajar con este ejemplo mortificándolo para mostrar las diferentes opciones que tenéis en Python:

- Declaramos la variable global **fuera del código** de los hilos principales y secundarios. Al inicio del código (CódigoEjemploA):

```python
import threading

# Variable global que comparten tanto el hilo principal como los hilos secundarios
contador_compartido = 0

def tarea():
    global contador_compartido
    contador = 0
    for _ in range(1000000):
        contador_compartido += 1
        contador +=1
    print("Soy el hilo:", threading.current_thread().name, ". El valor de mi contador es:", contador)

if __name__ == "__main__":

    hilos = []

    for i in range(1,10):
        # Crear dos hilos que comparten la variable global
        hilo = threading.Thread(target=tarea)
        hilos.append(hilo)
        hilo.name = "Hilo-"+str(i)
        hilo.start()

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

- Declaramos la variable global **en el hilo principal**. Nota que aquí (en hilo principal) no hace falta indicar que es global, pero en el hilo secundario sí es necesario indicarlo:
  - Dentro del hilo secundario, si no se pone que es una variable global, Python interpreta que es local (no se comparte con los otros hilos) y daría un error de no iniciada al querer sumarle i
  - CódigoEjemploB:

```python
import threading

def tarea():
    #Dentro de los hilos secundarios hay que indicar que la variable es global
    global contador_compartido
    contador = 0
    for _ in range(1000000):
        contador_compartido += 1
        contador +=1
    print("Soy el hilo:", threading.current_thread().name, ". El valor de mi contador es:", contador)

if __name__ == "__main__":

    # Variable global que comparten tanto el hilo principal como los hilos secundarios
    contador_compartido = 0

    hilos = []

    for i in range(1,10):
        # Crear dos hilos que comparten la variable global
        hilo = threading.Thread(target=tarea)
        hilos.append(hilo)
        hilo.name = "Hilo-"+str(i)
        hilo.start()

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

1.2. Escribir o leer una variable global, diferencias

Hay que diferencias bien entre acceder a una variable global sólo para consultarla (no hace falta ningún control) o acceder a una variable global para modificarla (hay que poner controles).

- **Lectura de Variables Globales**:

  Cuando múltiples hilos leen una variable global sin realizar modificaciones, no es necesario declarar la variable como global en el hilo. Python maneja automáticamente las lecturas de variables globales sin requerir una palabra clave adicional. De todas formas es aconsejable usar global para indicarlo.

  CódigoEjemploC:

```python
import threading
import time

def hilo_lectura():
    # Lectura de la variable global sin necesidad de 'global'. Aunque es aconsejable ponerlo
    valor = variable_global
    print(f"Hilo lector {threading.current_thread().name}: Leyendo variable global: {valor}")
    time.sleep(1)

def hilo_modificacion():
    # Modificación de la variable glogal, hay que indicar que es global
    global variable_global
    for _ in range(3):
        variable_global += 1
        print(f"Hilo modificador {threading.current_thread().name}: Modificando variable global a {variable_global}")
        time.sleep(1)

if __name__ == "__main__":
    # Variable global que comparten tanto el hilo principal como los hilos secundarios
    variable_global = 0

    hilosLectura = []
    hilosEscritura = []

    for i in range(1,5):
        # Crear dos hilos que comparten la variable global
        hilo = threading.Thread(target=hilo_lectura)
        hilosLectura.append(hilo)
        hilo.name = "HiloLectura-"+str(i)
        hilo.start()

        hilo = threading.Thread(target=hilo_modificacion)
        hilosEscritura.append(hilo)
        hilo.name = "HiloModif-"+str(i)
        hilo.start()


    print("Valor final de la variable global:", variable_global)
```

1.3. Resultados inesperados

Los resultados inesperados son problemas comunes en la programación concurrente y multihilo. Estos problemas surgen cuando dos o más hilos acceden y modifican compartidamente una variable sin una sincronización adecuada.

Para ilustrar mejor estos problema se muestra la salida de las ejecuciones de CódigoEjemploA, CódigoEjemploB y CódigoEjemploC.

- Ejecución del CódigoEjemploA:

```
Soy el hilo: Hilo-2 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-1 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-3 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-5 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-6 . El valor de mi contador es: 1000000
Valor final del contador compartido: 7388929
Soy el hilo: Hilo-8 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-4 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-7 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-9 . El valor de mi contador es: 1000000
```

```
Soy el hilo: Hilo-1 . El valor de mi contador es: 1000000
Valor final del contador compartido: 5865023
Soy el hilo: Hilo-6 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-8 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-2 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-5 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-4 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-3 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-7 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-9 . El valor de mi contador es: 1000000
```

  Como el hilo principal no espera hasta que acaben todos los hilos, imprime el total a mitad de las ejecuciones. En este ejemplo también se da la **condición de carrera** y **valores incorrectos**.

- Ejecución del CódigoEjemploC:

```
Hilo lector HiloLectura-1: Leyendo variable global: 0
Hilo modificador HiloModif-1: Modificando variable global a 1
Hilo lector HiloLectura-2: Leyendo variable global: 1
Hilo modificador HiloModif-2: Modificando variable global a 2
Hilo lector HiloLectura-3: Leyendo variable global: 2
Hilo modificador HiloModif-3: Modificando variable global a 3
Hilo lector HiloLectura-4: Leyendo variable global: 3
Hilo modificador HiloModif-4: Modificando variable global a 4
Valor final de la variable global: 4
Hilo modificador HiloModif-3: Modificando variable global a 5
Hilo modificador HiloModif-1: Modificando variable global a 6
Hilo modificador HiloModif-2: Modificando variable global a 7
Hilo modificador HiloModif-4: Modificando variable global a 8
Hilo modificador HiloModif-2: Modificando variable global a 9
Hilo modificador HiloModif-4: Modificando variable global a 10
Hilo modificador HiloModif-3: Modificando variable global a 11
Hilo modificador HiloModif-1: Modificando variable global a 12
```

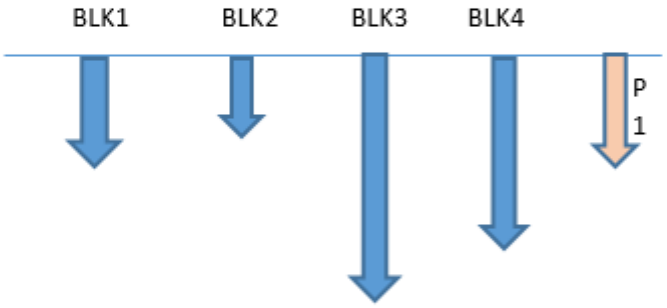
- **Resultados Inesperados:**
  - **Condición de Carrera:** Los hilos compiten por acceder y modificar la variable contador_compartido al mismo tiempo.

    En el ejemplo “CódigoEjemploA” varios hilos intentan incrementar la variable al mismo tiempo, el resultado final no será la multiplicación de los contadores de cada hilo. Esto se debe a que los pasos individuales de la operación += no están protegidos por un bloqueo, y los hilos pueden interferir entre sí, provocando resultados inesperados.
  - **Valores Incorrectos:** Como resultado de la condición de carrera, es probable que el valor final de contador_compartido sea menor de lo esperado, ya que algunos incrementos pueden perderse o sobrescribirse debido a la interferencia entre los hilos.
  - **Inconsistencia:** En diferentes ejecuciones, el valor final puede variar debido a la naturaleza no determinista de la concurrencia sin sincronización.

1.4. Join

Para solucionar una parte de los resultados inesperados vamos a introducir el método join de Python. Este método de la clase Thread se utiliza para esperar a que un hilo termine su ejecución antes de que el programa principal continúe. En otras palabras, si tienes un hilo en tu programa y quieres asegurarte de que el programa principal no avance hasta que ese hilo haya terminado, puedes usar el método join.

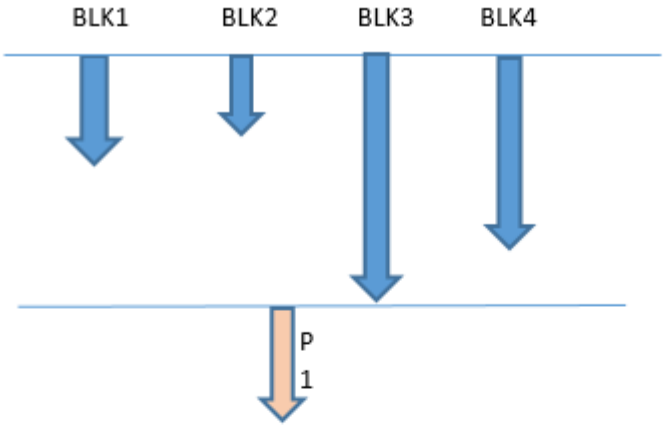
El “ CódigoEjemploA” ahora funciona de esta forma, todos los hilos se ejecutan al mismo tiempo:



By Sini Balakrishnan

https://vlsi.pro/system-verilog-fork-join/

Utilizando join tendíamos que el hilo principal (P1) espera a que todos los hilos secundarios acaben para luego imprimir el resultado de la variable:



By Sini Balakrishnan

https://vlsi.pro/system-verilog-fork-join/

La función join() se aplica a cada hilo secundario que queremos esperar. Si a un hilo secundario no se le llama a la función join, seguirá independientemente y el hilo principal no lo va a esperar.

Remarcar que en el momento en que se hace el join, a partir de esa línea es cuando se hace la espera, todo el código antes del join se ejecuta simultáneamente.

Veamos cómo queda el código modificado y el resultado:

```python
import threading

# Variable global que comparten tanto el hilo principal como los hilos secundarios
contador_compartido = 0

def tarea():
    global contador_compartido
    contador = 0
    for _ in range(1000000):
        contador_compartido += 1
        contador +=1
    print("Soy el hilo:", threading.current_thread().name, ". El valor de mi contador es:", con

if __name__ == "__main__":

    hilos = []

    for i in range(1,10):
        # Crear dos hilos que comparten la variable global
        hilo = threading.Thread(target=tarea)
        hilos.append(hilo)
        hilo.name = "Hilo-"+str(i)
        hilo.start()

    for hilo in hilos:
        hilo.join()


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

Resultado:

```
Soy el hilo: Hilo-2 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-3 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-1 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-8 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-5 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-6 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-4 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-9 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-7 . El valor de mi contador es: 1000000
Valor final del contador compartido: 3011930
```

Se resuelve el problema imprimir el valor total al final de la ejecución de todos los hilos secundarios, pero seguimos teniendo inconstencias.

Vemos que dada contador llega a 1.000.000. Hay 9 hilos que llegan a 1.000.000.

Si multiplicamos 9 * 1.000.000 = **9.000.000**
En cambio el total en este caso es **3.011.930**

He vuelto a ejecutar el código 3 veces más con los resultados:

- Valor final del contador compartido: **2.454.643**
- Valor final del contador compartido: **3.060.708**
- Valor final del contador compartido: **2.725.882**

¿Porqué, pasa esto?

Como hemos comentado con anterioridad, cuando se ejecutan varios hilos al mismo tiempo y comparten recursos o necesitan comunicarse entre ellos para coordinarse, pueden surgir problemas, y el resultado puede variar en cada ejecución. Por ello, la ejecución concurrente sin sincronización adecuada puede provocar resultados inesperados.

Soluciones que veremos a los problemas derivados de la Ejecución Concurrente:

- **Bloqueos**:

  Los bloqueos se utilizan para evitar el acceso simultáneo a un recurso compartido. Un hilo bloquea el recurso mientras lo está utilizando y lo libera cuando ha terminado.

- **Condiciones**:

  Similar a los bloqueos, las condiciones permiten a un hilo esperar hasta que se cumple cierta condición. Los hilos pueden notificar a otros cuando el recurso está disponible o cuando cierta condición se ha cumplido.

- **Semáforos**:

  Los semáforos son contadores que permiten controlar el acceso a un recurso. Pueden ser utilizados para limitar la cantidad de hilos que pueden acceder simultáneamente a un recurso compartido.

Estas técnicas de sincronización son herramientas esenciales para garantizar la coherencia y consistencia en programas multihilo. La elección de la técnica más adecuada dependerá de la naturaleza del problema y de los requisitos específicos de la aplicación.

2. Daemon

En el tema anterior hemos accedido a la variable name del constructor, para cambiar su valor:

```
hilo2 = threading.Thread(target=tarea_identificada, args=("rojo",), name="Hilo-B")
```

o

```
hilo2 = threading.Thread(target=tarea_identificada, args=("rojo",))
hilo2.name="Hilo-B"
```

La clase Thread también tiene una variable llamada Daemon que se utiliza para lanzar hilos en segundo plano.

Por un lado vamos ejecutando el hilo principal y comunicando valores al usuario y por detrás podríamos tener un hilo ejecutando por ejemplo ping a un servidor para asegurarnos que esta activo, recoger estadísticas o escuchar un evento del sistema para realizar una acción secundaria.

Es importante remarcar que tener un hilo daemon o no daemon influye en qué pasa cuando se termina la ejecución del hilo principal (programa principal):

- **Hilos no daemon** – varaible **Daemon = False**:
  - La configuración “no daemon” es el valor predeterminado. Todos los hilos tienen por defecto **Daemon = False**
  - El programa principal no terminará hasta que todos los hilos no daemon finalicen su ejecución. Dicho de otro modo, el hilo principal espera a que todos los hilos secundarios acaben antes de finalizar.
  - Se conoce como ejecución en **primer plano** o **Foreground**
- **Hilos daemon** – varaible **Daemon = True**:
  - Si un hilo se configura como "daemon" (**Daemon = True**), el programa principal **no espera** a que este hilo termine antes de salir. En el momento que el hilo principal acaba se detienen automáticamente todos los hilos “daemon”, sin importar si han terminado su trabajo o no.
  - Se conoce como ejecución en **segundo plano** o **Background**

Consideraciones importantes:

- Los hilos daemon son útiles cuando tienes tareas secundarias que no necesitas esperar a que terminen.
- Asegúrate de que un hilo daemon no esté realizando tareas críticas cuando el programa principal termine, ya que podría ser interrumpido abruptamente.
- Si todos los hilos en tu programa son daemon, el programa se detendrá tan pronto como todos los hilos no daemon hayan terminado.

Ejemplo de programa con hilos no daemon e hilos daemon:

```python
import threading
import time

def daemon_thread():
    while True:
        print("Hilo daemon ejecutándose...")
        time.sleep(1)

def non_daemon_thread():
    for i in range(5):
        print(f"Hilo no daemon ejecutándose ({i+1}/5)")
        time.sleep(2)

# Crear un hilo daemon
daemon_thread = threading.Thread(target=daemon_thread, daemon=True)

# Crear un hilo no daemon
non_daemon_thread = threading.Thread(target=non_daemon_thread, daemon=False)

# Iniciar los hilos
daemon_thread.start()
non_daemon_thread.start()

# Esperar a que el hilo no daemon termine antes de salir
non_daemon_thread.join()

print("Fin del programa")
```

La ejecución:

```
Hilo daemon ejecutándose...
Hilo no daemon ejecutándose (1/5)
Hilo daemon ejecutándose...
Hilo no daemon ejecutándose (2/5)
Hilo daemon ejecutándose...
Hilo daemon ejecutándose...
Hilo no daemon ejecutándose (3/5)
Hilo daemon ejecutándose...
Hilo daemon ejecutándose...
Hilo no daemon ejecutándose (4/5)
Hilo daemon ejecutándose...
Hilo daemon ejecutándose...
Hilo no daemon ejecutándose (5/5)
Hilo daemon ejecutándose...
Hilo daemon ejecutándose...
Fin del programa
```

Comentarios:

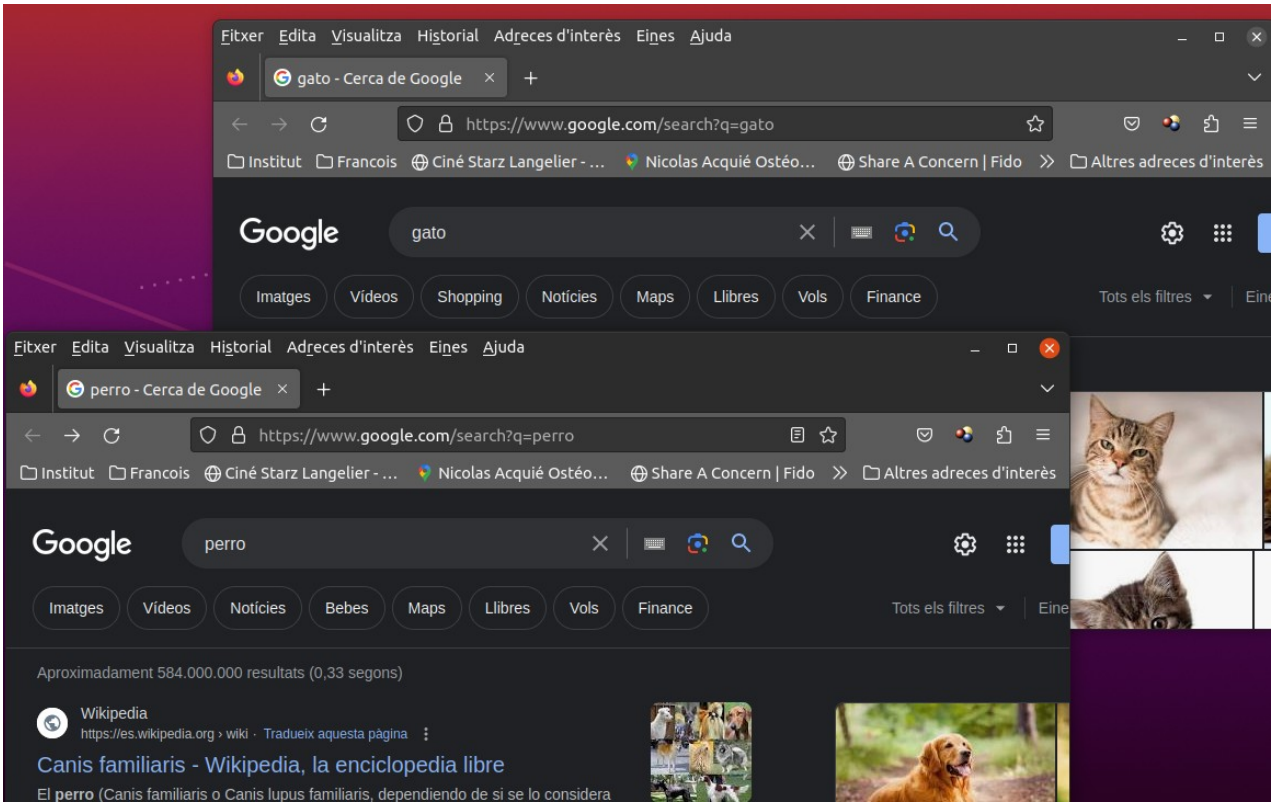
- El join() se usa para imprimir `print("Fin del programa")` al final del programa y no en cualquier lugar.
- Vemos que cuando el hilo **non_daemon_thread** llega a 2, finaliza y también finaliza el programa principal. Aunque el hilo daemon_thread seguiría imprimiendo, ya no puede, porque lo han finalizado de forma abrupta. Nadie espera a que acabe.

Otro ejemplo: Mientras voy haciendo ping a google (me aseguro que la página responde), le pido al usuario que me introduzca dos palabras a buscar. En este caso he pedido que busque: perros, gatos

```python
import threading
import time
import subprocess
from ping3 import ping, verbose_ping


def ping_google():
    while True:
        # Realizar ping a una página web (por ejemplo, google.com)
        result = ping("google.com")
        print(f"Ping a google.com: {result} ms")

        # Dormir durante un intervalo de tiempo (por ejemplo, 5 segundos)
        time.sleep(5)

def abrir_busqueda(thread_id):
    busqueda = input(f'Soy el thread {thread_id}. Me puedes indicar que quieres buscar?')
    # Dormir durante un intervalo de tiempo (por ejemplo, 3 segundos)
    time.sleep(3)

    try:
        # Especifica la búsqueda que quieres hacer
        url = "https://www.google.com/search?q="+busqueda

        # Utiliza subprocess.run para abrir Firefox con la URL
        subprocess.run(["firefox", url])

    except Exception as e:
        print(f"Error al abrir Firefox: {e}")


if __name__ == "__main__":
    # Crear un hilo daemon para el ping
    ping_thread = threading.Thread(target=ping_google, daemon=True)

    hilos = []

    #Creamos 5 hilos que pediran buscar algo en google. Como el parámatreo por defecto es
    #daemon=False, no lo indicamos.
    for i in range (1,3):
        hilo_busca = threading.Thread(target=abrir_busqueda, args=(i,))
        hilos.append(hilo_busca)
        hilo_busca.start()

    # Iniciar también el hilo daemon
    ping_thread.start()

    # El programa principal puede continuar realizando otras tareas
    for _ in range(2):
        print("Programa principal realizando otras tareas...")
        time.sleep(4)

    # Esperar a que los hilos no daemon terminen (opcional)
    for hilo in hilos:
        hilo.join()


    #El programa principal espera a que acaben los hilos no daemon.
    #Imprime un mensaje final:

    print("Programa principal terminado (el hilo de ping daemon continuará en segundo plano).")
```

El resultado de ejecutar el código (os lo dejo en los anexos):

```
Soy el thread 1. Me puedes indicar que quieres buscar?Programa principal realizando otras tareas...
Ping a google.com: 0.051870107650756836 ms
Programa principal realizando otras tareas...
Ping a google.com: 0.0991520881652832 ms
gato
Soy el thread 2. Me puedes indicar que quieres buscar?Ping a google.com: 0.17730498313903809 ms
perro
Ping a google.com: 0.1810743808746338 ms
Programa principal terminado (el hilo de ping daemon continuará en segundo plano).
```



3. Herencia de la Clase Thread en Python

La herencia en la programación orientada a objetos es un concepto fundamental que permite la creación de clases secundarias que heredan atributos y métodos de una clase principal. En el contexto de la programación multihilo en Python, esta herencia se extiende a menudo a la clase `Thread`, permitiendo la personalización de los hilos con funcionalidades adicionales.

Hasta ahora, el método `start()` se utilizaba para establecer los parámetros y datos con los que se lanzaría la ejecución del hilo, utilizando el parámetro `args` y otros. Al modificar la clase `Thread`, se logra una mayor flexibilidad para añadir propiedades y métodos adicionales, lo que simplifica el código del programa principal o hilo principal. Esta modificación permite delegar parte del trabajo y estructura a la propia clase, promoviendo así la reutilización del código.

Ejemplo de Herencia en la Clase Thread:

```python
import threading
import time

class MiHilo(threading.Thread):
    def __init__(self, nombre, contador, intervalo):
        super().__init__(name=nombre)
        self.contador = contador
        self.intervalo = intervalo

    def run(self):
        print(f"{self.name} iniciado")
        for i in range(1, self.contador + 1):
            print(f"{self.name}: Contador {i}")
            time.sleep(self.intervalo)
        print(f"{self.name} completado")

if __name__ == "__main__":
    # Crear instancias de la clase hija (MiHiloMejorado)
    hilo1 = MiHilo(nombre="Hilo 1", contador=3, intervalo=1)
    hilo2 = MiHilo(nombre="Hilo 2", contador=5, intervalo=0.5)

    # Iniciar los hilos
    hilo1.start()
    hilo2.start()

    # Esperar a que ambos hilos terminen
    hilo1.join()
    hilo2.join()

    print("Programa principal terminado")
```

En este ejemplo, hemos añadido dos propiedades adicionales a la clase MiHilo: contador e intervalo. Estos atributos son configurados a través del constructor __init__.

Al modificar __init__ hemos modificado el constructor. Podemos ver el cambio que hay al instanciar un objeto de la clase Thread:

- Antes (sin herencia de clase):
  - hilo1 = threading.Thread(target=funcion_contador, args=(contador=3, intervalo=1))
- Ahora (con herencia de clase):
  - hilo1 = MiHilo(nombre="Hilo 1", contador=3, intervalo=1)

La función run de la clase Thread también ha sido adaptada para imprimir el contador y pausar según el intervalo especificado.

De esta manera, cada instancia de MiHilo puede tener su propio contador y intervalo de tiempo entre impresiones. Con la herencia onseguimos tenemos mayor flexibilidad para añadir propiedades y métodos, que nos pueden ayudar a simplificar el código del programa o hilo principal, dado que parte de ese trabajo y estructura se lo pasamos a la clase.

4. Sincronismo

Hemos comentado con anterioridad, que al ejecutar múltiples tareas simultáneamente, surge la necesidad crítica de sincronizar estas operaciones para garantizar un comportamiento coherente y predecible del programa. El sincronismo se convierte en una pieza fundamental en la construcción de sistemas robustos, donde hilos y procesos comparten recursos y trabajan en armonía.

El sincronismo aborda los desafíos inherentes a la concurrencia, tales como las condiciones de carrera, donde múltiples hilos compiten por acceder a un recurso compartido, y los resultados inesperados que pueden surgir cuando las operaciones no están cuidadosamente coordinadas. La sincronización se convierte, entonces, en la clave para prevenir conflictos y garantizar la consistencia en la ejecución de programas multitarea.

4.1. El Problema de los Jardines

Tenemos un parque con dos accesos y queremos controlar el acceso de los visitantes (no queremos que haya más de 200 personas en el parque).

Se desea poder conocer en cualquier momento el número de visitantes a los jardines, por lo que se dispone de un ordenador con conexión en cada puerta.

Cada puerta informa al ordenador si se produce una entrada o una salida.

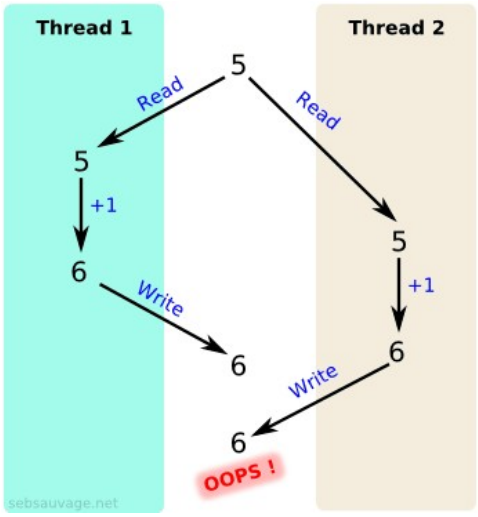
Programación:

- El hilo principal crea 2 hilos secundarios:
  - P1: hilo que se asigna a una puerta
  - P2: hilo que se asigna a la otra puerta
- Ambos hilos se ejecutan de forma concurrente y utilizan una única variable x para llevar la cuenta del número de visitantes.
- El incremento o decremento de la variable se produce cada vez que un visitante entra o sale por una de las puertas.
  - Entrada de un visitante por una de las puertas → x:=x+1
  - Salida de un visitante por una de las puertas → x:=x-1

Si ambas instrucciones se realizaran como una única instrucción hardware, no se plantearía ningún problema y el programa podría funcionar siempre correctamente.

El problema es que la actualización de la variable x se realiza mediante la ejecución de otras instrucciones más sencillas, como son:

- Copiar el valor de x en un registro del procesador.
- Incrementar el valor del registro
- Almacenar el resultado en la dirección donde se guarda x



https://sebsauvage.net/python/snyppets/

- Supongamos **x=5**
- Si en la **entrada 1** (**Thread 1**) entra una persona:
  - copiar el valor de x en un registro del procesador (valor 5).
  - incrementar el valor del registro (valor 6)
- **Cambio de contexto**
- En la **entrada 2** (**Thread 2**) entra una persona:
  - copiar el valor de x en un registro del procesador (valor 5).
  - incrementar el valor del registro (valor 6)
  - almacenar el resultado en la dirección donde se guarda x (valor 6)
- **Cambio de contexto**
- El Thread 1 sigue con la ejecución de su código:
  - almacenar el resultado en la dirección donde se guarda x (valor 6)

Podemos ver que han entrado dos personas pero una se ha convertido en un “fantasma” y no la hemos contado.

Para solucionar este problema Python nos ofrece el Bloqueo o Lock

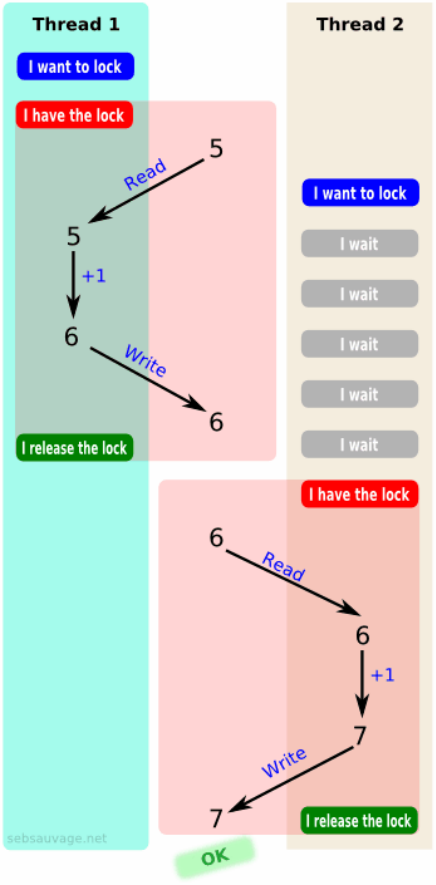
4.2. El objeto lock en Python

El objeto Lock en Python es una herramienta de sincronización proporcionada por el módulo threading que se utiliza para garantizar la exclusión mutua entre hilos. Exclusión mutua significa que solo un hilo puede acceder a una sección crítica del código a la vez. El propósito principal de un Lock es prevenir las condiciones de carrera y garantizar la coherencia de los datos compartidos entre hilos.

Los métodos clave del objeto Lock son:

- **Adquisición (acquire)**: Cuando un hilo adquiere un Lock, bloquea la ejecución de otros hilos hasta que lo libera.
- **Liberación (release)**: Cuando un hilo libera un Lock, permitiendo d e esta forma que otros hilos lo puedan adquirir.

Esto asegura que las secciones críticas del código protegidas por el Lock se ejecuten de manera exclusiva.



https://sebsauvage.net/python/snyppets/

Hay que controlar los errores, es por eso que se aconseja el uso de with. Se podría hacer con try … except, pero con with es una forma más sencilla y simple de garantizar garantiza que el Lock se libera incluso si ocurren excepciones dentro de la sección crítica.

Para ver cómo funciona usaremos el ejercicio " CódigoEjemploA" con las funciones: lock.acquire() y lock.release()

```python
import threading

# Variable global que comparten tanto el hilo principal como los hilos secundarios
contador_compartido = 0

# Lock para sincronización
lock = threading.Lock()

def tarea():
    global contador_compartido
    contador = 0
    for _ in range(1000000):
        # Adquirir el bloqueo antes de modificar la variable compartida
        try:
            lock.acquire()
            contador_compartido += 1
        except Exception as e:
            print(f"Error al incrementar el contador: {e}")
        finally:
            # Siempre liberar el bloqueo en el bloque finally
            lock.release()
        # Las variables locales no hace falta protegerlas
        contador +=1
    print("Soy el hilo:", threading.current_thread().name, ". El valor de mi contador es:", contador)

if __name__ == "__main__":
    hilos = []

    for i in range(1,10):
        # Crear dos hilos que comparten la variable global
        hilo = threading.Thread(target=tarea)
        hilos.append(hilo)
        hilo.name = "Hilo-"+str(i)
        hilo.start()

    for hilo in hilos:
        hilo.join()

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

En este código, `lock.acquire()` se coloca en un bloque `try`, y `lock.release()` se coloca en un bloque `finally`. Esto asegura que el bloqueo se libere incluso si ocurre una excepción dentro del bloque `try`. Utilizar `try`, `except`, y `finally` de esta manera es una práctica común para garantizar la liberación adecuada de recursos, como bloqueos.

Simplificando el código anterior con with tendríamos:

```python
import threading

# Variable global que comparten tanto el hilo principal como los hilos secundarios
contador_compartido = 0

# Lock para sincronización
lock = threading.Lock()

def tarea():
    global contador_compartido
    contador = 0
    for _ in range(1000000):
        # Adquirir el bloqueo antes de modificar la variable compartida
        with lock:
            contador_compartido += 1
        # Las variables locales no hace falta protegerlas
        contador +=1
    print("Soy el hilo:", threading.current_thread().name, ". El valor de mi contador es:", contador)

if __name__ == "__main__":

    hilos = []

    for i in range(1,10):
        # Crear dos hilos que comparten la variable global
        hilo = threading.Thread(target=tarea)
        hilos.append(hilo)
        hilo.name = "Hilo-"+str(i)
        hilo.start()

    for hilo in hilos:
        hilo.join()


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

La salida de este código es:

```
Soy el hilo: Hilo-1 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-2 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-6 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-7 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-9 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-5 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-8 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-3 . El valor de mi contador es: 1000000
Soy el hilo: Hilo-4 . El valor de mi contador es: 1000000
Valor final del contador compartido: 9000000
```

Gracias a que hemos controlado la región crítica, ahora cada vez que ejecutemos este código obtendremos el mismo resultado final del contador: **9.000.000**

El uso incorrecto de bloqueos (Locks) puede llevar a problemas como bloqueos infinitos, inanición y bloqueos de la aplicación si no se manejan adecuadamente. Aquí hay algunos problemas comunes relacionados con el uso de bloqueos y qué puede suceder si no se liberan correctamente:

- **Bloqueo Infinito**:
  - Si un hilo adquiere un bloqueo y no lo libera adecuadamente, otros hilos pueden quedar bloqueados indefinidamente esperando ese bloqueo.
  - Esto puede ocurrir si hay un error o excepción en la sección crítica y el bloqueo no se libera en la cláusula finally.
- **Inanición (Starvation**):
  - Un problema menos obvio es la inanición, que ocurre cuando un hilo o un conjunto de hilos siempre son incapaces de adquirir un bloqueo debido a la competencia con otros hilos.
  - Esto puede suceder si no se manejan adecuadamente las prioridades o si un hilo siempre adquiere el bloqueo antes que otros.

Es por ello crucial liberar los bloqueos de manera adecuada para evitar estos problemas. Utilizar un bloque try, except, y finally es una buena práctica ya que asegura que el bloqueo se libere incluso si ocurren errores.

Te todas formas es mucho **más aconsejable usar with** en estos casos, dado que aseguras de forma automática la liberación del bloqueo, incluso si ocurren excepciones dentro del bloque. El uso de with es especialmente valioso en entornos complejos de concurrencia para evitar problemas como bloqueos infinitos y condiciones de carrera.

4.3. Combinar la herencia de clases y el uso de lock

En la programación concurrente, la combinación de la herencia de clases y el uso de bloqueos se convierte en un enfoque poderoso para gestionar la ejecución de múltiples hilos de manera segura y eficiente. La herencia de la clase Thread en Python facilita la creación de hilos, mientras que los bloqueos (Locks) proporcionan un mecanismo robusto para sincronizar el acceso a recursos compartidos. En esta sección, exploraremos cómo estructurar clases que heredan de Thread y utilizan bloqueos para garantizar la integridad de los datos en entornos multitarea.

Presentaremos un ejemplo concreto que demuestra este enfoque, mostrando cómo un bloqueo puede incorporarse en la inicialización de un hilo para gestionar una sección crítica compartida.

```python
import threading

# Variable global que comparten tanto el hilo principal como los hilos sec
contador_compartido = 0

class ContadorThread(threading.Thread):
    def __init__(self,lock):
        super().__init__()
        self.lock = lock


    def run(self):
        global contador_compartido
        contador = 0
        for _ in range(1000):
            # Adquirir el bloqueo antes de modificar la variable compartid
            with self.lock:
                contador_compartido += 1
            # Las variables locales no hace falta protegerlas
            contador +=1
        print("Soy el hilo:", threading.current_thread().name, ". El valor

if __name__ == "__main__":

    # Lock para sincronización de todos los threads.
    # Un único Lock para todos
    lock = threading.Lock()

    hilos = []

    for i in range(1,10):
        # Crear dos hilos que comparten la variable global
        # A los hilos se le pasa la misma referencia al mismo objeto lock
        hilo = ContadorThread(lock)
        hilos.append(hilo)
        hilo.name = "Hilo-"+str(i)
        hilo.start()

    for hilo in hilos:
        hilo.join()

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

5. Anexo: Los objetos y sus funciones en Python. Diferencias con Java

Vamos a ver un poco las diferencias que tiene Java y Python de definir las clases.

En Java, no se utiliza this de manera explícita al definir métodos dentro de una clase, pero se usa this de manera implícita para hacer referencia a los campos y métodos de la instancia actual.

5.1. Definición de Clases en Java:

En Java, la definición de una clase se realiza de la siguiente manera:

```java
public class MiClase {
    private int miCampo;

    public void setMiCampo(int valor) {
        // En Java, el compilador utiliza 'this' implícitamente
        this.miCampo = valor; // 'this' se usa implícitamente
    }
}
```

En Java, cuando defines un método dentro de una clase, no es necesario utilizar la palabra clave this.

Se utiliza this de manera implícita para referirse a la instancia actual (this.miCampo).

5.2. Definición de Clases en Python:

En Python, la definición de una clase se realiza de la siguiente manera:

```python
class MiClase:
    def __init__(self):
        self.mi_campo = 0

    def set_mi_campo(self, valor):
        # En Python, 'self' se usa explícitamente
        self.mi_campo = valor
```

En Python, se utiliza self de manera explícita al definir métodos de clase para hacer referencia a la instancia actual. Por ello se utiliza la palabra clave `self` como el primer parámetro de los métodos de clase (incluido el constructor `__init__`).

Aunque se podría llamar a este primer parámetro de cualquier manera (no es un requerimiento llamarlo self, pero es una convención ampliamente aceptada), es una buena práctica mantenerlo como self para mejorar la legibilidad y la consistencia del código.

Al igual que en java (this), se utiliza self para acceder a los campos y métodos de la instancia dentro de los métodos de la clase.

¿Por qué se usa `self` en Python?

La razón fundamental para utilizar `self` en Python es indicar explícitamente a qué instancia de la clase pertenecen los campos y métodos. En Python, no se usa `this` como en Java, ya que Python prefiere la legibilidad y la simplicidad. Al utilizar `self`, queda claro que estás haciendo referencia a los atributos o métodos de la instancia actual de la clase. Esto mejora la comprensión del código y evita posibles ambigüedades.

Además, Python valora la legibilidad y la simplicidad en su diseño, y `self` se ajusta a esta filosofía al hacer que el código sea más claro y explícito en cuanto a las relaciones entre los métodos y atributos de una clase y sus instancias.