Instituto Tecnológico de Colima

Daniel Hernández Pizano
Maestría en Sistemas Computacionales
Unidad II Concurrencia de Hilos
Práctica Concurrencia en Python (segunda oportunidad)
DanielPizanoo/Practica-Concurrencia (github.com)

Introducción

Python incluye herramientas sofisticadas para el manejo de operaciones concurrentes a través de la utilización de procesos e hilos de ejecución. Incluso muchos programas relativamente simples pueden hacerse ejecutar más rápido aplicando técnicas para ejecutar partes del trabajo al mismo tiempo utilizando módulos.

Con la concurrencia las computadoras hacen que funcionen aún más rápido, a través de la concurrencia, podemos lograr esto y nuestros programas de Python podrán manejar aún más solicitudes a la vez, y con el tiempo, lo que generará impresionantes ganancias de rendimiento. En informática, la concurrencia es la ejecución de trabajos o tareas por una computadora al mismo tiempo. Normalmente, una computadora ejecuta un trabajo mientras otros esperan su turno, una vez que se completa, los recursos se liberan y el siguiente trabajo comienza a ejecutarse. Este no es el caso cuando se implementa la simultaneidad, ya que las piezas de trabajo a ejecutar no siempre tienen que esperar a que se completen otras. Se ejecutan al mismo tiempo.

Un hilo es la unidad de ejecución mas pequeña que se puede realizar en una computadora. Los subprocesos existen como partes de un proceso y, por lo general, no son independientes entre sí, lo que significa que comparten datos y memoria con otros subprocesos dentro del mismo proceso. Los subprocesos tambien se denominan a veces procesos ligeros [1].

En Python un objeto **Thread** representa una determinada operación que se ejecuta como un subproceso independiente, es decir, es la representación de un hilo [2].

El módulo **`threading`** incluye un API de alto nivel orientada a objetos para el trabajo con concurrencia en Python. Los objetos **`Thread`** corren concurrentemente dentro del mismo proceso y comparten la memoria. La utilización de hilos es una manera sencilla para escalar tareas que están más vinculadas con operaciones de I/O que de CPU.

La manera más sencilla para utilizar un objeto **`Thread`**, es instanciar éste con una función objetivo y llamar el método **`start()`** para permitir que éste comience a trabajar. Realice en Python el siguiente script:

```
import threading

def trabajador():
    """funcion del hilo"""
```

```
    print('Trabajador')

hilos = []
for i in range(5):
    h = threading.Thread(target=trabajador)
    hilos.append(h)
    h.start()
```

Es útil poder generar un hilo y pasarle argumentos que le indiquen qué hacer; para esto, cualquier tipo de objeto puede ser pasado al hilo. Por ejemplo, el siguiente código pasa un número que luego imprime el hilo.

```
import threading

def trabajador(num):
    """funcion del hilo"""
    print('Trabajador: %s' % num)

hilos = []
for i in range(5):
    h = threading.Thread(target=trabajador, args=(i,))
    hilos.append(h)
    h.start()
```

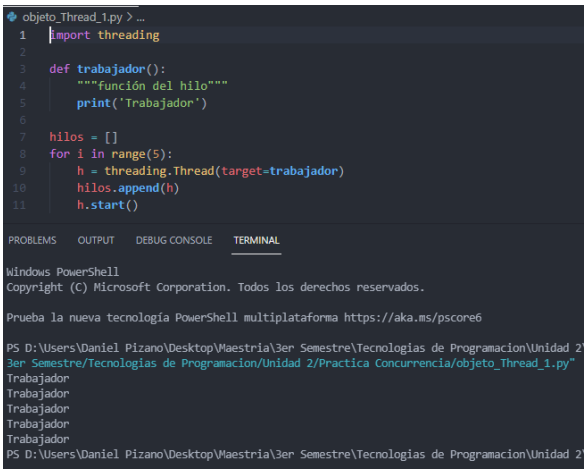
Tomando en cuenta los ejemplos anteriores, en el segundo se puede observar que se le concateno un contador, el cual mientras vaya recorriendo el ciclo for del tamaño del rango otorgado del hilo, este se va a ir incrementando y se mostrará junto al texto.
En el segundo script se puede observar el texto definido como “args”, el cual permite pasar un número variable de argumentos a una función, por lo que si se quiere definir una función cuyo número de parámetros de entrada puede ser variable, se considera el uso de args como una opción. El nombre de args viene de argumentos, que es como se denomina en programación a los parámetros de entrada de una función [3].
Un ejemplo del funcionamiento de args seria:

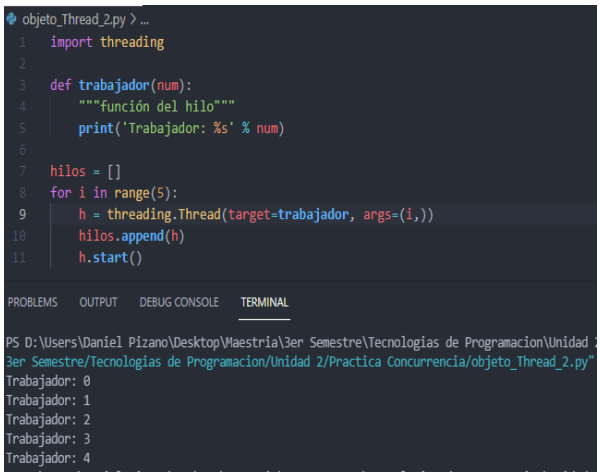
```
def test_var_args(f_args, *argv):
        print(“primer argumento normal:”, f_arg)
        for arg in argv:
                print(“argumentos de *argv:”, arg)

test_var_args(‘python’, ‘foo’, ‘bar’)
```

Y la salida que produce el código anterior al llamarlo con 3 parámetros es la siguiente:

```
primer argumento normal: python
argumentos de *argv: foo
argumentos de *argv: bar
```

En todo caso, es mejor crear un hilo pasando como argumento el método que ejecutará la funcionalidad del hilo, ya que se obtendrá un código más limpio, porque queda más claro, en este caso, cual es el trabajador. En lugar de mantener el código dentro de una subclase Thread, la cual se pueda utilizar el código de la función trabajador en otras partes del código.





Determinar el hilo actual.

Usar argumentos para identificar y nombrar un hilo es innecesario, ya que cada instancia **Thread** tiene un nombre con un valor por defecto que puede ser cambiado cuando el hilo es creado. Nombrar hilos es útil en procesos servidor en dónde múltiples hilos de servicio manejan diferentes operaciones.

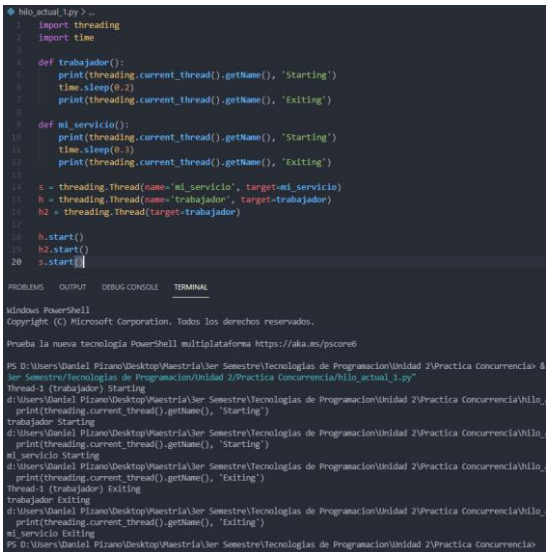
```
import threading
import time

def trabajador():
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(0.2)
    print(threading.current_thread().getName(), 'Exiting')

def mi_servicio():
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(0.3)
    print(threading.current_thread().getName(), 'Exiting')

s = threading.Thread(name='mi_servicio', target=mi_servicio)
h = threading.Thread(name='trabajador', target=trabajador)
h2 = threading.Thread(target=trabajador)

h.start()
h2.start()
s.start()
```

En la ejecución del script anterior se muestran los tres nombres de los hilos, y los tres mensajes de Starting y Exiting, los cuales corresponden a cada uno de los hilos creados cuando se ejecutan los métodos que se les asignó al parámetro target. El hilo h2 aparece con el nombre de Thread-1 porque no se le asignó un nombre específico a través de los parámetros name al instanciar Thread, al cual se le asignó ese nombre genérico. Si se les quita esos parámetros a las otras dos instancias, ocurre lo mismo y son nombradas de acuerdo al orden en que fueron instanciadas.

Muchos programas no usan el **`print`** para depuración; por lo cual, el módulo **`logging`** soporta incluir el nombre del hilo en cada mensaje log usando el **`%(threadName)s`** `formateador de código`. Incluir el nombre de los hilos en los mensajes log hace posible rastrear hacia atrás el origen de los mensajes.

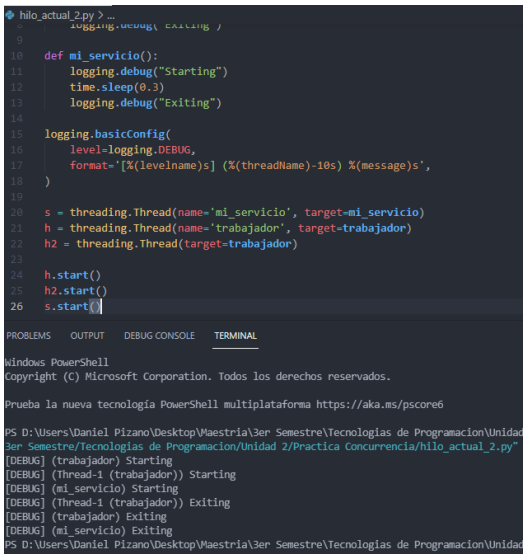
```
import threading
import time
import logging

def trabajador():
    logging.debug("Starting")
    time.sleep(0.2)
    logging.debug("Exiting")

def mi_servicio():
    logging.debug("Starting")
    time.sleep(0.3)
    logging.debug("Exiting")

logging.basicConfig(
    level=logging.DEBUG,
    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
)

s = threading.Thread(name='mi_servicio', target=mi_servicio)
h = threading.Thread(name='trabajador', target=trabajador)
h2 = threading.Thread(target=trabajador)

h.start()
h2.start()
s.start()
```

Los formateadores tienen los siguientes atributos y métodos. Son responsables de convertir un LogRecord en una cadena que puede ser interpretada por un humano o un sistema externo. El formateador base permite especificar una cada de formato. Si no se proporciona ninguno, '%(Message)s' es usado. El formateador tiene varias llaves. Se utiliza para referirse a la información a visualizar, en este caso comienza con '%(levelname)s', indica el nombre del nivel del registro, que puede ser las siguientes ('DEBUG','INFO','WARNING','ERROR','CRITICAL'), luego, %(ThreadName)-10s. Obtener el nombre del hilo %(message)s, obtiene el mensaje pasado durante la llamada logging.debug() [4].



```
import threading
import time

def trabajador():
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(0.2)
    print(threading.current_thread().getName(), 'Exiting')

def mi_servicio():
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(0.3)
    print(threading.current_thread().getName(), 'Exiting')

s = threading.Thread(name='mi_servicio', target=mi_servicio)
h = threading.Thread(name='trabajador', target=trabajador)
h2 = threading.Thread(target=trabajador)

h.start()
h2.start()
s.start()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS D:\Users\Daniel Pizano\Desktop\Maestria\3er Semestre\Tecnologias de Programacion\Unidad 2\Practica Concurrencia> &
3er Semestre/Tecnologias de Programacion/Unidad 2/Practica Concurrencia/hilo_actual_1.py"
Thread-1 (trabajador) Starting
d:\Users\Daniel Pizano\Desktop\Maestria\3er Semestre\Tecnologias de Programacion\Unidad 2\Practica Concurrencia\hilo_
  print(threading.current_thread().getName(), 'Starting')
trabajador Starting
d:\Users\Daniel Pizano\Desktop\Maestria\3er Semestre\Tecnologias de Programacion\Unidad 2\Practica Concurrencia\hilo_
  print(threading.current_thread().getName(), 'Starting')
mi_servicio Starting
d:\Users\Daniel Pizano\Desktop\Maestria\3er Semestre\Tecnologias de Programacion\Unidad 2\Practica Concurrencia\hilo_
  print(threading.current_thread().getName(), 'Exiting')
Thread-1 (trabajador) Exiting
trabajador Exiting
d:\Users\Daniel Pizano\Desktop\Maestria\3er Semestre\Tecnologias de Programacion\Unidad 2\Practica Concurrencia\hilo_
  print(threading.current_thread().getName(), 'Exiting')
mi_servicio Exiting
PS D:\Users\Daniel Pizano\Desktop\Maestria\3er Semestre\Tecnologias de Programacion\Unidad 2\Practica Concurrencia>
```



```
def mi_servicio():
    logging.debug("Starting")
    time.sleep(0.3)
    logging.debug("Exiting")

logging.basicConfig(
    level=logging.DEBUG,
    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
)

s = threading.Thread(name='mi_servicio', target=mi_servicio)
h = threading.Thread(name='trabajador', target=trabajador)
h2 = threading.Thread(target=trabajador)

h.start()
h2.start()
s.start()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS D:\Users\Daniel Pizano\Desktop\Maestria\3er Semestre\Tecnologias de Programacion\Unidad
3er Semestre/Tecnologias de Programacion/Unidad 2/Practica Concurrencia/hilo_actual_2.py"
[DEBUG] (trabajador) Starting
[DEBUG] (Thread-1 (trabajador)) Starting
[DEBUG] (mi_servicio) Starting
[DEBUG] (Thread-1 (trabajador)) Exiting
[DEBUG] (trabajador) Exiting
[DEBUG] (mi_servicio) Exiting
PS D:\Users\Daniel Pizano\Desktop\Maestria\3er Semestre\Tecnologias de Programacion\Unidad
```

Hilos tipo demonio

Hasta ahora todos los programas han implícitamente esperado a terminar hasta que todos los hilos han completado su trabajo. Algunas veces, sin embargo, los programas generan un hilo como un ***demonio*** que corre sin bloquear el programa principal de su salida. Los hilos tipo demonio, son útiles para servicios puede que no haya una manera fácil de interrumpir el hilo, o donde dejar que el hilo muera en medio de su trabajo no conduce a la pérdida o corrupción de datos. Para marcar el hilo como un demonio, se debe pasar el parámetro **`daemon = True`** cuando se construye o llamar su método **`set_daemon()`** con **`True`**.

```
import threading
import time
import logging

def demonio():
    logging.debug('Starting')
    time.sleep(0.2)
    logging.debug('Exiting')

def no_demonio():
    logging.debug('Starting')
    logging.debug('Exiting')

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

d = threading.Thread(name='daemon', target=demonio, daemon=True)
h = threading.Thread(name='non-daemon', target=no_demonio)

d.start()
h.start()
```

Los subprocesos que siempre se ejecutan en segundo plano brindan soporte a subprocesos principales o no demonio, esos subprocesos que se ejecutan en segundo plano se consideran subprocesos daemon. El hilo de utilidad no bloquea el flujo principal que sale y continúa funcionando en el fondo [5].
La función principal de Daemon es proporcionar servicios convenientes para la ejecución de otros subprocesos. La aplicación más típica del subproceso es el GC (recolector de basura), que es un tutor competente [6].

Modifique el script anterior para utilizar el método **`join().`**

```
import threading
import time
import logging

def demonio():
    logging.debug('Starting')
    time.sleep(0.2)
    logging.debug('Exiting')

def no_demonio():
    logging.debug('Starting')
    logging.debug('Exiting')

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

d = threading.Thread(name='daemon', target=demonio, daemon=True)
h = threading.Thread(name='non-daemon', target=no_demonio)
d.start()
h.start()

d.join()
h.join()
```

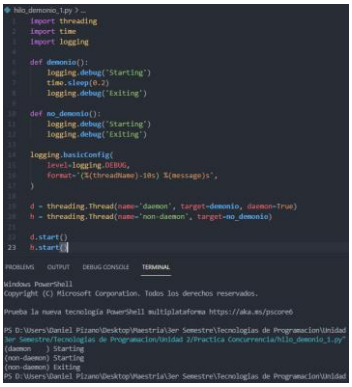
En el script anterior, se muestra el mensaje de Exiting del hilo daemon, esto sucede porque al usar el join(), el hilo daemon puede finalizar la ejecución del hilo antes de finalizar el código.

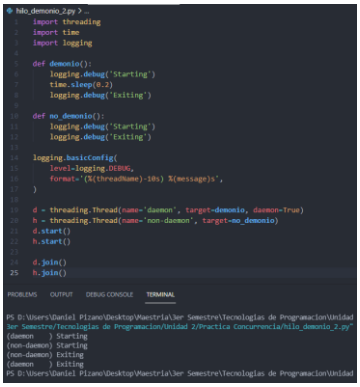
Alternativamente, un valor flotante puede ser pasado para representar el número de segundos para esperar a que el hilo se convierta en inactivo.

```
import threading
import time
import logging


def demonio():
    logging.debug('Starting')
    time.sleep(0.2)
    logging.debug('Exiting')
```

```python
def no_demonio():
    logging.debug('Starting')
    logging.debug('Exiting')


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

d = threading.Thread(name='daemon', target=demonio, daemon=True)
h = threading.Thread(name='non-daemon', target=no_demonio)
d.start()
h.start()

d.join(0.1)
print('d.isAlive()', d.is_alive())
h.join()
```

En este script, al final de la ejecución del código, se muestra un mensaje que indica que el hilo d, todavía está durante la ejecución, cuando se pasa un limite de tiempo en el método join(), indica cuánto tiempo lleva antes de continuar ejecutándose el código, debe esperar a que se complete el hilo, como se muestra en este ejemplo. El tiempo de espera es menor que el tiempo de espera encontrado en la función Daemon(), el hilo no tiene tiempo para terminar, por lo que no se muestra mensaje de salida.





Enumeración de hilos.

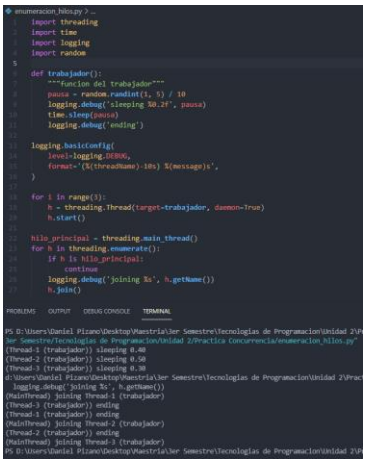
No es necesario mantener un manejador explícito para todos los hilos de tipo demonio para asegurar que han terminado después de que el proceso principal sale; para esto, el método enumerate() regresa una lista de instancias activas de Thread.

```python
import logging

def trabajador():
    """funcion del trabajador"""
    pausa = random.randint(1, 5) / 10
    logging.debug('sleeping %0.2f', pausa)
    time.sleep(pausa)
    logging.debug('ending')
```

```
logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

for i in range(3):
    h = threading.Thread(target=trabajador, daemon=True)
    h.start()

hilo_principal = threading.main_thread()
for h in threading.enumerate():
    if h is hilo_principal:
        continue
    logging.debug('joining %s', h.getName())
    h.join()
```

La función threading.enumerate(), retorna una lista de todos los objetos tipo Thread actualmente con vida. La lista incluye hilos demonio, objetos hilo dummy creados por current_thread(), y el hilo principal. Excluye hilos terminados e hilos que todavía no hayan sido iniciados [7].

En el script anterior, existe una función trabajador() que muestra dos mensajes, de suspensión y salida. El formato del mensaje de registro también está configurado. Luego, a través de for, Thread se instancia 4 veces y el método de destino apunta a la función trabajador(), finalmente en otro for se recorre la lista de los hilos para mostrar el mensaje joining, el cual se utiliza para no imprimir el mensaje joining con el hilo principal, el cual es el hilo con el que el intérprete de Python fue inicializado.



Señalización entre eventos.

Aun cuando el punto de utilizar múltiples hilos es correr operaciones separadas concurrentemente, algunas veces es importante ser capaz de sincronizar las operaciones en uno o más hilos. Los objetos **`Event`** son una manera sencilla de comunicarse de manera segura entre procesos. Un **`Event`** maneja una señal interna que los llamantes pueden controlar con los métodos **`set()`** y **`clear()`**. Otros hilos pueden utilizar **`wait()`** para pausar

hasta que la bandera es fijada, bloqueando efectivamente el progreso hasta que a aquellos hilos se les permite continuar.

```
import logging
import threading
import time

def espera_por_evento(e):
    """Wait for the event to be set before doing anything"""
    logging.debug('espera_por_evento iniciando')
    estableciendo_evento = e.wait()
    logging.debug('estableciendo evento: %s', estableciendo_evento)

def espera_por_evento_timeout(e, t):
    """Wait t seconds and then timeout"""
    while not e.is_set():
        logging.debug('espera_por_evento_timeout iniciando')
        estableciendo_evento = e.wait(t)
        logging.debug('estableciendo evento: %s', estableciendo_evento)
        if estableciendo_evento:
            logging.debug('procesando evento')
        else:
            logging.debug('haciendo otro trabajo')

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

e = threading.Event()

t1 = threading.Thread(name='block',target=espera_por_evento,args=(e,),)
t1.start()

t2 =
threading.Thread(name='nonblock',target=espera_por_evento_timeout,args=(e
, 2),)
t2.start()

logging.debug('Esperando antes de llamar el Event.set()')
time.sleep(0.3)
e.set()
logging.debug('El evento es establecido')
```

El objeto de la clase Event proporciona un mecanismo simple que se utiliza para la comunicación entre subprocesos donde un subproceso señala un evento mientras los otros subprocesos lo esperan. Entonces, cuando un hilo que está destinado a producir la señal la produce, entonces se activa el hilo en espera. El objeto de evento conocido como bandera de evento usa una bandera interna que se puede establecer como verdadera usando el método set() y se puede restablecer a falsa usando el método clear(). El método wait() bloquea un hilo hasta que el indicador de evento que está esperando, sea establecido como verdadero por cualquier otro hilo [8].

En este script, espera_por_evento_timeout() comprueba el status del evento sin bloqueo indefinido. El espera_por_evento() bloquea en la llamada a wait(), que no regresa hasta que el estado del evento cambie.

Control de Acceso a Recursos.

En adición a los eventos otra manera de sincronizar hilos es a través del uso de condiciones a través de un objeto **`Condition`**. Debido a que el objeto **`Condition`** usa un **`Lock`**, este puede ser ligado a un recurso compartido, permitiendo que múltiples hilos esperen por el recurso a ser actualizado. En el siguiente ejemplo el hilo **`consumidor()`** espera que se establezca la condición antes de continuar. El hilo **`productor()`** es responsable de establecer la condición y notificar a los otros hilos que ellos pueden continuar.

```
import logging
import threading
import time

def consumidor(cond):
    """espera por la condición y utilizar el recurso"""
    logging.debug('Iniciando hilo consumidor')
    with cond:
        cond.wait()
        logging.debug('Recurso disponible para el consumidor')

def productor(cond):
    """coloca el recurso a ser utilizado por el consumidor"""
    logging.debug('Iniciando hilo productor')
    with cond:
        logging.debug('Haciendo el recurso disponible')
        cond.notifyAll()

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s (%(threadName)-2s) %(message)s',
)
```

```
condicion = threading.Condition()
c1 = threading.Thread(name='c1', target=consumidor,args=(condicion,))
c2 = threading.Thread(name='c2', target=consumidor,args=(condicion,))
p = threading.Thread(name='p', target=productor,args=(condicion,))

c1.start()
time.sleep(0.2)
c2.start()
time.sleep(0.2)
p.start()
```

La clase threading.condition() implementa objetos de condición variable. Una condición variable permite que uno o más hilos esperen hasta que sean notificados por otro hilo. Si se provee un argumento lock distinto de none, debe ser un objeto Lock o RLock, y se utiliza como el lock subyacente. De otro modo, se crea un nuevo objeto RLock y se utiliza como el lock subyacente [7]. Con la instrucción with, se asegura que el comando cond se encuentre disponible al utilizarlo en el código que contiene el with.

En adición a la sincronización de operaciones de los hilos, es importante ser capaz de controlar el acceso a recursos compartidos para prevenir la corrupción o pérdida de los datos. Los tipos de datos de Python como listas, diccionarios son atómicos por lo cual un solo hilo a la vez puede acceder a ellos. Otros tipos de datos simples implementados en Python como enteros o flotantes no tienen esta protección. Para proteger el acceso simultáneo a un objeto se utiliza el objeto **Lock,** con la finalidad de lograr la ***exclusión mutua***.

```
import logging
import random
import threading
import time

class Contador:
    def __init__(self, start=0):
        self.lock = threading.Lock()
        self.valor = start

    def incrementar(self):
        logging.debug('Esperando por lock')
        self.lock.acquire()

        try:
            logging.debug('Lock adquirido')
            self.valor = self.valor + 1
        finally:
            logging.debug('Lock liberado')
            self.lock.release()

def trabajador(c):
    for i in range(2):
        pausa = random.random()
        logging.debug('Durmiendo %0.02f', pausa)
        time.sleep(pausa)
```

```
        c.incrementar()
    logging.debug('Done')

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

contador = Contador()
for i in range(2):
    t = threading.Thread(target=trabajador, args=(contador,))
    t.start()

logging.debug('Esperando por hilos trabajador ')
main_thread = threading.main_thread()

for t in threading.enumerate():
    if t is not main_thread:
        t.join()
logging.debug('Contador: %d', contador.valor)
```

La clase que implemente los objetos de la primitiva lock. Una vez que un hilo ha adquirido un lock, intentos subsecuentes por adquirirlo bloquearán, hasta que sea liberado; cualquier hilo puede liberarlo. Un lock reentrante es una primitiva de sincronización que puede ser adquirida múltiples veces por el mismo hilo. Internamente, utiliza el concepto de «hilo dueño» y «nivel de recursividad» además del estado abierto/cerrado utilizado por las primitivas locks. Si está en estado cerrado, algún hilo es dueño del lock; si está en estado abierto, ningún hilo es dueño. Para cerrar el lock, un hilo llama a su método acquire(); esto retorna una vez que el hilo se ha adueñado del lock. Para abrir el lock, un hilo llama a su método release(). Pares de llamadas acquire()/release() pueden anidarse; sólo el release() final (el release() del par más externo) restablece el lock ha abierto y permite que otro hilo bloqueado en acquire() proceda [7].

En el script anterior, la clase Lock se usa varias veces en el hilo de trabajo, pero debido a que el hilo está bloqueado, esperará hasta que se libere el bloqueo. El hilo de trabajo toma la función de trabajo como objetivo, enviar la instancia de la clase de contado como parámetro, imprime el mensaje de suspensión en la función de trabajo, espera a dormir() y luego llama a la clase de método increment() del contador. La clase de contado en el método increment() es el lugar donde se bloquea el hilo, se incrementa la variable de valor y se libera el hilo.

control_acceso_recursos_2.py > ...

```python
import logging
import random
import threading
import time

class Contador:
    def __init__(self, start=0):
        self.lock = threading.Lock()
        self.valor = start

    def incrementar(self):
        logging.debug('Esperando por lock')
        self.lock.acquire()

        try:
            logging.debug('Lock adquirido')
            self.valor = self.valor + 1
        finally:
            logging.debug('Lock liberado')
            self.lock.release()

def trabajador(c):
    for i in range(2):
        pausa = random.random()
        logging.debug('Durmiendo %0.02f', pausa)
        time.sleep(pausa)
        c.incrementar()
    logging.debug('Done')

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

contador = Contador()
for i in range(2):
    t = threading.Thread(target=trabajador, args=(contador,))
    t.start()

logging.debug('Esperando por hilos trabajador ')
main_thread = threading.main_thread()

for t in threading.enumerate():
    if t is not main_thread:
        t.join()
logging.debug('Contador: %d', contador.valor)
```

control_acceso_recursos_2.py > ...

```python
import logging
import random
import threading
import time

class Contador:
    def __init__(self, start=0):
        self.lock = threading.Lock()
        self.valor = start

    def incrementar(self):
        logging.debug('Esperando por lock')
        self.lock.acquire()

        try:
            logging.debug('Lock adquirido')
            self.valor = self.valor + 1
        finally:
            logging.debug('Lock liberado')
            self.lock.release()
```


PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
3er Semestre/Tecnologias de Programacion/Unidad 2/Practica Concurrencia/control_acceso_recursos_2.py"
(Thread-1 (trabajador)) Durmiendo 0.92
(Thread-2 (trabajador)) Durmiendo 0.85
(MainThread) Esperando por hilos trabajador
(Thread-2 (trabajador)) Esperando por lock
(Thread-2 (trabajador)) Lock adquirido
(Thread-2 (trabajador)) Lock liberado
(Thread-2 (trabajador)) Durmiendo 0.92
(Thread-1 (trabajador)) Esperando por lock
(Thread-1 (trabajador)) Lock adquirido
(Thread-1 (trabajador)) Lock liberado
(Thread-1 (trabajador)) Durmiendo 0.74
(Thread-1 (trabajador)) Esperando por lock
(Thread-1 (trabajador)) Lock adquirido
(Thread-1 (trabajador)) Lock liberado
(Thread-1 (trabajador)) Done
(Thread-2 (trabajador)) Esperando por lock
(Thread-2 (trabajador)) Lock adquirido
(Thread-2 (trabajador)) Lock liberado
(Thread-2 (trabajador)) Done
(MainThread) Contador: 4
PS D:\Users\Daniel Pizano\Desktop\Maestria\3er Semestre\Tecnologias de Programacion\Unidad 2\Practica
```

Referencias

[1] de la Vega, R. (2021, 29 enero). ⊳ Concurrencia en Python. Pharos. https://pharos.sh/concurrencia-en-python/

[2] Fernández, J. C. (2017, 17 agosto). Curso Python. Volumen XX: Hilos (Threading). Parte I. RedesZone. https://www.redeszone.net/2017/07/13/curso-python-volumen-xx-hilos-parte-i/

[3] 1. Uso de *args y **kwargs — documentación de Python Intermedio - 0.1. (s. f.). Python Intermedio. https://python-intermedio.readthedocs.io/es/latest/args_and_kwargs.html

[4] 6.29.6 Formatter Objects. (s. f.). Referencia de la biblioteca de Python. https://docs.python.org/2.4/lib/node357.html

[5] Hilos del demonio de Python – Acervo Lima. (s. f.). ACERVO LIMA. https://es.acervolima.com/hilos-del-demonio-de-python/

[6] Resumen de hilos de daemon en Java - programador clic. (s. f.). programador clic. https://programmerclick.com/article/6315309926/

[7] threading — Paralelismo basado en hilos — documentación de Python - 3.8.12. (s. f.). Python. https://docs.python.org/es/3.8/library/threading.html

[8] Python Event Object | Studytonight. (s. f.). Study Tonight. https://www.studytonight.com/python/python-threading-event-object

[9] Schmidt, E. R. (2019). threading — Gestionar operaciones concurrentes dentro de un proceso. El módulo Python 3 de la semana. https://rico-schmidt.name/pymotw-3/threading/