

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

Guía para el tercer laboratorio de
INF239 Sistemas Operativos

Tema: Concurrencia en Python

1. Definiciones importantes

A continuación se presenta dos conceptos importantes que suelen confundirse mucho: concurrencia y paralelismo.

“Traditionally, the word *parallel* is used for systems in which the executions of several programs overlap in time by running them on separate processors. The word *concurrent* is reserved for potential parallelism, in which the executions may, but need not, overlap; instead, the parallelism may only be apparent since it may be implemented by sharing the resources of a small number of processors, often only one. Concurrency is an extremely useful *abstraction* because we can better understand such a program by pretending that all processes are being executed in parallel. Conversely, even if the processes of a concurrent program are actually executed in parallel on several processors, understanding its behavior is greatly facilitated if we impose an order on the instructions that is compatible with shared execution on a single processor. Like any abstraction, concurrent programming is important because the behavior of a wide range of real systems can be modeled and studied without unnecessary detail.”

Principles of Concurrent and Distributed Programming, Second Edition by M. Ben-Ari

“We define a **concurrent** program as one in which multiple streams of instructions are active at the same time. One or more of the streams is available to make progress in a single unit of time. The key to differentiating parallelism from concurrency is the fact that through time-slicing or multitasking one can give the illusion of simultaneous execution when in fact only one stream makes progress at any given time.

In systems where we have multiple processing units that can perform operations at the exact same time, we are able to have instruction streams that execute in parallel. The term parallel refers to the fact that each stream not only has an abstract timeline that executes concurrently with others, but these timelines are in reality occurring simultaneously instead of as an illusion of simultaneous execution based on interleaving within a single timeline. A concurrent system only ensures that two sequences of operations may appear to happen at the same time when examined in a coarse time scale (such as a user watching the execution) even though they may do so only as an illusion through rapid interleaving at the hardware level, while a parallel system may actually execute them at the same time in reality.

Our definition of a **parallel** program is an instance of a concurrent program that executes in the presence of multiple hardware units that will guarantee that two or more instruction streams will make progress in a single unit of time. The differentiating factor from a concurrent program executing via time-slicing is that in a parallel program, at least two streams make progress at a given time. This is most often a direct consequence of the presence of multiple hardware units that can support simultaneous execution of distinct streams.”

Introduction Concurrency in Programming Languages by Matthew J. Sottile, Timothy G. Mattson, Craig E Rasmussen

“Concurrency vs. parallelism

- Concurrency is about dealing with lots of things at once.
- Parallelism is about doing lots of things at once.
- Not the same, but related.
- Concurrency is about structure, parallelism is about execution.
- Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.”

Rob Pike

Le invitamos a escuchar la siguiente conferencia de Rob Pike: “[Concurrency Is Not Parallelism](#)”

2.- ¿ Qué problemas puede surgir cuando hay concurrencia ?

Básicamente hay dos grandes grupos en los que caen todos los problemas. El primero es protección del recurso compartido, y el segundo es sincronización.

Recurso compartido, en estos casos hay un recurso, que puede ser memoria compartida, las variables globales cuando se trata de hilos, o archivos tal como las bases de datos. El problema radica en que para mantener la integridad del recurso, solo un proceso debe acceder a este (exclusión mutua). Un ejemplo clásico que se usa, en los medios académicos, es el de la variable compartida. Hay varios hilos, cada uno de ellos lo incrementa una cantidad fija de veces. Al final el total debe ser igual a la suma de los incrementos de cada hilo. Sin embargo esto a veces no se llega a cumplir.

Sincronización, hay situaciones donde la multitarea (o paralelismo) puede beneficiarnos frente a una programación secuencial. Imagínese la siguiente situación: se tiene una fábrica de gaseosa donde se llena la botella y por otro lado se fabrica la tapa. Llenar 1000 botellas toma 20 minutos, fabricar 1000 tapas toma 10 minutos. Si se ejecuta de forma secuencial, todo tomaría 30 minutos. Sin embargo se puede ejecutar de forma paralela, la fabricación de las tapas y la del llenado de la botella. El problema es que si primero acaba el que fabrica la tapa, el brazo que coloca la tapa en la botella, tiene que esperar a que haya una botella llena. A esto se llama sincronización. Aún así, el tiempo que se gana es considerable.

Todos estos problemas surgen porque los procesos (o hilos) acaban de forma no determinista, compitiendo unos con otros. Por ese motivo a este tipo de problemas también se les denomina ***Race condition***.

En estos casos su tarea será asegurar la exclusión mutua al recurso compartido si este presenta problemas de *race condition*. O sincronizar los procesos para que cumplan un flujo específico. En ambos casos deberá conseguir concurrencia lo más que se pueda. Si anula por completo la concurrencia, significa que finalmente se están ejecutando secuencialmente. Y esto no es una solución.

3. Thread-based parallelism (Python 3.8)

El presente material pretende mostrar con algunos ejemplos el uso de las herramientas de concurrencia que proporciona Python. No pretende explicar cómo funcionan, en Internet, usted puede encontrar la documentación correspondiente.

La documentación se encuentra en: <https://docs.python.org/3/library/threading.html>

Todos los ejemplos serán escritos como *scripts* y la ejecución siempre será desde la terminal.

4.- Thread Objects

Hilos en Python

Un hilo, también llamado proceso ligero, es un flujo de ejecución diferente al hilo principal. Todo proceso tiene un hilo principal de ejecución y a partir de ahí se pueden crear diferentes hilos. Estos hilos, son planificados por la librería o por el kernel, dependiendo del tipo de hilos que son (hilos software o hilos kernel).

Está claro que cuando uno crea hilos, es por que desea tener varios flujos de ejecución sobre un mismo código o sobre código diferente. Es decir, un parámetro obligatorio al momento de crear un hilo será el nombre de la función (código) que se desea que se ejecute.

Empezaremos importando el módulo que permite usar hilos y escribiendo una función que será la que ejecute los hilos.

Antes de continuar, es necesario resaltar que muchas IDE's para Python tienen incorporado en una ventana el *shell* de Python, para que el resultado de la ejecución se muestre en ella, sin necesidad de salir de la IDE. Esta forma de trabajar, sin embargo puede alterar el resultado obtenido de la ejecución del programa. Por este motivo mostraremos la ejecución en una terminal en lugar de la IDE.

```
ej1.py x
1  import threading
2
3  def sumador(x,y):
4      print(x+y)
5      return
6
7  t = threading.Thread(target=sumador,args=(4,5,))
8  t.start()
9
```

```
alejandro@abdebien:ThreadsBasico$ python3 ej1.py
9
alejandro@abdebien:ThreadsBasico$ █
```

Nota: En la mayoría de los casos la salida de la IDE y la del terminal coinciden. Cada IDE tiene un botón que ejecuta el interprete sobre el código y muestra el resultado.

Obviamente uno podría crear más de un hilo.

```
ej1.py x ej2.py x
1  import threading
2
3  def sumador(x,y):
4      print(x+y)
5      return
6
7  t1 = threading.Thread(target=sumador,args=(1,2,))
8  t2 = threading.Thread(target=sumador,args=(3,4,))
9  t3 = threading.Thread(target=sumador,args=(5,6,))
10 t4 = threading.Thread(target=sumador,args=(7,8,))
11 t1.start()
12 t2.start()
13 t3.start()
14 t4.start()
```

```
alejandro@abdebien:ThreadsBasico$ python3 ej2.py
3
7
11
15
alejandro@abdebien:ThreadsBasico$ █
```

O mejor al estilo Python

```
ej1.py x ej2.py x ej3.py x
1  import threading
2
3  def sumador(x,y):
4      print(x+y)
5      return
6
7  for par in [(1,2),(3,4),(5,6),(7,8)]:
8      t = threading.Thread(target=sumador,args=(par[0],par[1],))
9      t.start()
```

```
alejandro@abdebien:ThreadsBasico$ python3 ej3.py
3
7
11
15
alejandro@abdebien:ThreadsBasico$ █
```

Importante: cuando los hilos son planificados, la secuencia en que son ejecutados dependerá del planificador (librería o kernel). En nuestro ejemplo, debido a que el código del hilo es muy pequeño han sido ejecutados en el mismo orden en que han sido creados. Si el código hubiera sido un poco más largo es posible que el planificador hubiera decidido ejecutar en otro orden en una segunda ronda.

Importante: en Python el hilo principal siempre espera a que terminen cualquier otro hilo que haya sido creado (esto se puede comprobar observando cómo está escrito el módulo). Si uno explícitamente desea indicar que el hilo principal espere para unirse al hilo creado, debería usar el método *join*.

Otra forma de crear y ejecutar hilos es usando el módulo *concurrent.futures*. Este módulo nos permite crear muchos hilos en una línea.

```
ej1.py  ej2.py  ej3.py  ej4.py x  ej5.py *
1  import concurrent.futures
2
3  def sumador(x,y):
4      print(x+y)
5      return
6
7  with concurrent.futures.ThreadPoolExecutor(max_workers=4) as executor:
8      for par in [(1,2),(3,4),(5,6),(7,8)]:
9          executor.submit(sumador,par[0],par[1])
10
11
```

```
alejandro@abdebien:ThreadsBasico$ python3 ej4.py
3
7
11
15
alejandro@abdebien:ThreadsBasico$
```

Race condition

A continuación mostraremos un problema de concurrencia. Primero crearemos un objeto de la clase *Myclass* con solo una variable *self._var* y el método *incrementa* que lo único que hace es incrementar en 100 000 la variable *self._var*. Luego crearemos un *pool* de 100 hilos que ejecutan este método y finalmente mostraremos el valor de la variable *self._var*. Nosotros deberíamos esperar que su valor sea 10 000 000. La lógica nos indica que si el método lo incrementa en 100 000 y hay 100 hilos que ejecutan el método, en total el valor debería ser 10 000 000.

```
ej1.py  ej2.py  ej3.py  ej4.py  ej5.py x
1  import concurrent.futures
2  import time
3
4  class Myclass:
5
6      def __init__(self):
7          self._var = 0
8
9      def incrementa(self):
10         for _ in range(100000):
11             temporal = self._var
12             temporal = temporal + 1
13             self._var = temporal
14
15     def get(self):
16         return self._var
17
18 myobject = Myclass()
19 with concurrent.futures.ThreadPoolExecutor(max_workers=100) as executor:
20     futures = {executor.submit(myobject.incrementa) for _ in range(100)}
21     concurrent.futures.wait(futures)
22
23 print(myobject.get())
24
```

Sin embargo la salida por terminal no muestra siempre el mismo resultado.

```
alejandro@abdebien:ThreadsBasico$ python3 ej5.py
8480402
alejandro@abdebien:ThreadsBasico$ python3 ej5.py
8310713
alejandro@abdebien:ThreadsBasico$ python3 ej5.py
8375306
alejandro@abdebien:ThreadsBasico$ python3 ej5.py
8315195
alejandro@abdebien:ThreadsBasico$ python3 ej5.py
8354772
alejandro@abdebien:ThreadsBasico$ python3 ej5.py
8789039
alejandro@abdebien:ThreadsBasico$ █
```

La explicación es muy sencilla aunque será expuesta con detalle en clase, en esta guía se darán algunas pinceladas.

Cuando se planifica un hilo se le asigna un tiempo de ejecución (*quantum*). Si hay varios hilos y si el tiempo de total de cada uno de ellos es mayor al *quantum*, este se va rotando para cada hilo según la política del planificador. Es decir la ejecución de cada hilo se corta en alguna línea del código. Esto impide que la actualización de la variable se complete y de allí que la variable tenga valores incorrectos.

En Python existen muchas herramientas para sincronizar hilos, en este caso emplearemos **Lock Objects**

```
ej1.py  ej2.py  ej3.py  ej4.py  ej5.py  ej6.py x
1  import concurrent.futures
2  import threading
3  import time
4
5  class Myclass:
6
7      def __init__(self):
8          self._var = 0
9          self._lock = threading.Lock()
10
11     def incrementa(self):
12         for _ in range(100000):
13             with self._lock:
14                 temporal = self._var
15                 temporal = temporal + 1
16                 self._var = temporal
17
18
19     def get(self):
20         return self._var
21
22 myobject = Myclass()
23 with concurrent.futures.ThreadPoolExecutor(max_workers=100) as executor:
24     futures = {executor.submit(myobject.incrementa) for _ in range(100)}
25     concurrent.futures.wait(futures)
26
27 print(myobject.get())
```

```

alejandro@abdebian:ThreadsBasico$ python3 ej6.py
10000000
alejandro@abdebian:ThreadsBasico$ python3 ej6.py
10000000
alejandro@abdebian:ThreadsBasico$ python3 ej6.py
10000000
alejandro@abdebian:ThreadsBasico$ █

```

A continuación se presentan problemas y en cada caso se resuelven empleando un objeto proporcionado por Python.

Problema 1

```

problemal.py
1 import threading as th
2
3 def hilo(num):
4     print(f"Hi, congratulations you are running your first threading program, i am thread {num}.")
5     print(f"Python threads are used in cases where the execution of a task involves some waiting, i am thread {num}.")
6     print(f"Threading allows python to execute other code while waiting, i am thread {num}.")
7
8 hilos = []
9 for n in range(4):
10     h = th.Thread(target=hilo,args=(n,))
11     hilos.append(h)
12
13 for n in range(4):
14     hilos[n].start()
15

```

```

alejandro@abdebian:Concurrencia$ python3 problemal.py
Hi, congratulations you are running your first threading program, i am thread 0.
Python threads are used in cases where the execution of a task involves some waiting, i am thread 0.
Hi, congratulations you are running your first threading program, i am thread 1.
Threading allows python to execute other code while waiting, i am thread 0.
Hi, congratulations you are running your first threading program, i am thread 2.
Python threads are used in cases where the execution of a task involves some waiting, i am thread 2.
Python threads are used in cases where the execution of a task involves some waiting, i am thread 1.
Threading allows python to execute other code while waiting, i am thread 1.
Hi, congratulations you are running your first threading program, i am thread 3.
Threading allows python to execute other code while waiting, i am thread 2.
Python threads are used in cases where the execution of a task involves some waiting, i am thread 3.
Threading allows python to execute other code while waiting, i am thread 3.
alejandro@abdebian:Concurrencia$ █

```

Se crean 4 hilos (de 0 a 3) y cada uno de ellos debe de imprimir 3 líneas de texto. Se puede observar en la salida que esto no sucede. Cada hilo, al imprimir su mensaje, ha sido interrumpido por los otros hilos. Esto es por que, la pantalla también es un recurso compartido. Lo interesante es que esta situación no es constante, usted puede ejecutar varias veces y obtendrá distintos resultados.

Lo que se desea en este caso es que cada hilo imprima su mensaje completo (las tres líneas) antes de que otro hilo haga uso de la pantalla.

Solución 1 – Otra forma de emplear Lock Objects

```
problema1.py  solucion1.py x
1  import threading as th
2
3  def hilo(num):
4      l.acquire()
5      print(f"Hi, congratulations you are running your first threading program, i am thread {num}.")
6      print(f"Python threads are used in cases where the execution of a task involves some waiting, i am thread {num}.")
7      print(f"Threading allows python to execute other code while waiting, i am thread {num}.")
8      l.release()
9
10 hilos = []
11 l = th.Lock()
12 for n in range(4):
13     h = th.Thread(target=hilo,args=(n,))
14     hilos.append(h)
15
16 for n in range(4):
17     hilos[n].start()
```

```
alejandro@abdebian:Concurrencia$ python3 solucion1.py
Hi, congratulations you are running your first threading program, i am thread 0.
Python threads are used in cases where the execution of a task involves some waiting, i am thread 0.
Threading allows python to execute other code while waiting, i am thread 0.
Hi, congratulations you are running your first threading program, i am thread 1.
Python threads are used in cases where the execution of a task involves some waiting, i am thread 1.
Threading allows python to execute other code while waiting, i am thread 1.
Hi, congratulations you are running your first threading program, i am thread 2.
Python threads are used in cases where the execution of a task involves some waiting, i am thread 2.
Threading allows python to execute other code while waiting, i am thread 2.
Hi, congratulations you are running your first threading program, i am thread 3.
Python threads are used in cases where the execution of a task involves some waiting, i am thread 3.
Threading allows python to execute other code while waiting, i am thread 3.
alejandro@abdebian:Concurrencia$ █
```

Observe ahora la salida, ningún hilo es interrumpido. Todos imprimen su mensaje antes que otro lo haga. Al ejecutarlo varias veces, siempre obtendrá el mismo resultado

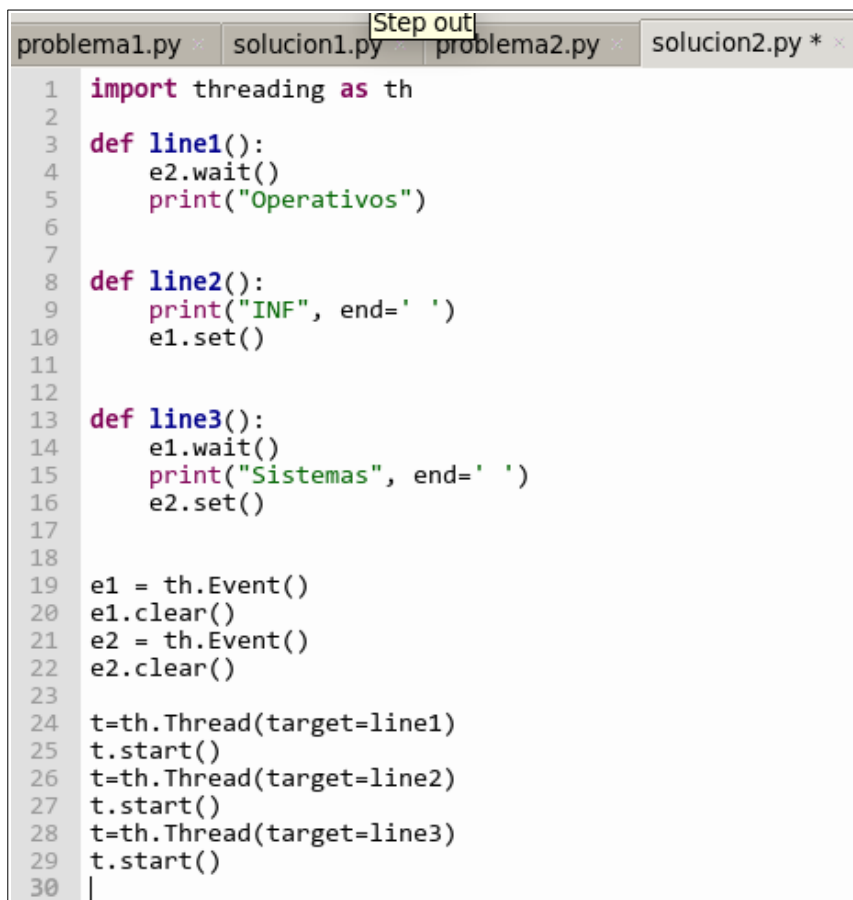
Problema 2

```
problema1.py  solucion1.py  problema2.py x
1  import threading as th
2
3  def line1():
4      print("Operativos")
5
6  def line2():
7      print("INF", end=' ')
8
9  def line3():
10     print("Sistemas", end=' ')
11
12 t=th.Thread(target=line1)
13 t.start()
14 t=th.Thread(target=line2)
15 t.start()
16 t=th.Thread(target=line3)
17 t.start()
```

```
alejandro@abdebian:Concurrencia$ python3 problema2.py
Operativos
INF Sistemas alejandro@abdebian:Concurrencia$ █
```

El objetivo es sincronizar los tres hilos para que impriman: **INF Sistemas Operativos**

Solución 2 - Empleando Event Objects



```
1 import threading as th
2
3 def line1():
4     e2.wait()
5     print("Operativos")
6
7
8 def line2():
9     print("INF", end=' ')
10    e1.set()
11
12
13 def line3():
14     e1.wait()
15     print("Sistemas", end=' ')
16     e2.set()
17
18
19 e1 = th.Event()
20 e1.clear()
21 e2 = th.Event()
22 e2.clear()
23
24 t=th.Thread(target=line1)
25 t.start()
26 t=th.Thread(target=line2)
27 t.start()
28 t=th.Thread(target=line3)
29 t.start()
30 |
```



```
alejandro@abdebien:Concurrencia$ python3 solucion2.py
INF Sistemas Operativos
alejandro@abdebien:Concurrencia$ █
```

Problema 3

El objetivo es simular el problema del productor-consumidor usando *buffer* limitado. En Python los arreglos no están definidos nativamente, pero pueden ser sustituidos por listas. Una lista de Python puede ser accedida por un índice, como en los arreglos. Una lista puede crecer si se emplean los métodos `insert` y `append`. No hay otra forma de modificar su tamaño. Nosotros definiremos su tamaño al inicio y no usaremos los métodos mencionados, de esta forma se comportará como un arreglo de tamaño estático. Por este motivo protegeremos el acceso a la lista, exigiendo que el índice esté entre los límites de la lista.

El comportamiento del productor-consumidor debe ser el siguiente: siempre que haya espacio el productor puede producir un elemento y almacenarlo en el *buffer*. Cuando el *buffer* se encuentra lleno debe esperar. Por otro lado el consumidor debe consumir elementos del *buffer*, siempre y cuando haya elementos que consumir, en caso contrario debe de esperar.

problema1.py	solucion1.py	problema2.py	solucion2.py	problema3.py
--------------	--------------	--------------	--------------	--------------

```

1 import threading as th
2
3 indice = -1
4 buffer = [None, None, None, None, None]
5
6 def productor():
7     for n in range(20):
8         global indice
9         global buffer
10        item = n*n
11        if indice < 5:
12            indice += 1
13            buffer[indice]=item
14            print(f"productor {indice} {item} {buffer}", flush=True)
15
16 def consumidor():
17     item = None
18     for _ in range(20):
19         global indice
20         global buffer
21         if indice > -1:
22             item = buffer[indice]
23             buffer[indice]=None
24             print(f"consumidor {indice} {item} {buffer}", flush=True)
25             indice -= 1
26
27 t=th.Thread(target=productor)
28 t.start()
29 t=th.Thread(target=consumidor)
30 t.start()
31
32

```

```

alejandro@abdebian:Concurrencia$ python3 problema3.py
productor 0 0 [0, None, None, None, None]
productor 1 1 [None, 1, None, None, None]
consumidor 0 0 [None, None, None, None, None]
productor 2 4 [None, 1, 4, None, None]
productor 2 9 [None, None, 9, None, None]
consumidor 1 1 [None, None, 4, None, None]
productor 3 16 [None, None, 9, 16, None]
consumidor 2 9 [None, None, None, 16, None]
productor 3 25 [None, None, None, 25, None]
consumidor 2 None [None, None, None, 25, None]
productor 3 36 [None, None, None, 36, None]
consumidor 2 None [None, None, None, 36, None]
productor 3 49 [None, None, None, 49, None]
consumidor 2 None [None, None, None, 49, None]
productor 3 64 [None, None, None, 64, None]
consumidor 2 None [None, None, None, 64, None]
productor 3 81 [None, None, None, 81, None]
consumidor 2 None [None, None, None, 81, None]
productor 3 100 [None, None, None, 100, None]
consumidor 2 None [None, None, None, 100, None]
productor 3 121 [None, None, None, 121, None]
consumidor 2 None [None, None, None, 121, None]
productor 3 144 [None, None, None, 144, None]
consumidor 2 None [None, None, None, 144, None]
productor 3 169 [None, None, None, 169, None]
consumidor 2 None [None, None, None, 169, None]
productor 3 196 [None, None, None, 196, None]
consumidor 2 None [None, None, None, 196, None]
productor 3 225 [None, None, None, 225, None]
consumidor 2 None [None, None, None, 225, None]
productor 3 256 [None, None, None, 256, None]
consumidor 2 None [None, None, None, 256, None]
productor 3 289 [None, None, None, 289, None]
consumidor 2 None [None, None, None, 289, None]
productor 3 324 [None, None, None, 324, None]
consumidor 2 None [None, None, None, 324, None]
productor 3 361 [None, None, None, 361, None]
consumidor 2 None [None, None, None, 361, None]
consumidor 1 None [None, None, None, 361, None]
consumidor 0 None [None, None, None, 361, None]
alejandro@abdebian:Concurrencia$ █

```

```

alejandro@abdebian:Concurrencia$ python3 problema3.py
productor 0 0 [0, None, None, None, None]
productor 1 1 [0, 1, None, None, None]
productor 2 4 [0, None, 4, None, None]
productor 3 9 [0, None, 4, 9, None]
productor 4 16 [0, None, 4, 9, 16]
consumidor 1 1 [0, None, None, None, None]
consumidor 4 16 [0, None, 4, 9, None]
consumidor 3 9 [0, None, 4, None, None]
consumidor 2 4 [0, None, None, None, None]
consumidor 1 None [0, None, None, None, None]
consumidor 0 0 [None, None, None, None, None]
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/usr/lib/python3.7/threading.py", line 917, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.7/threading.py", line 865, in run
    self._target(*self._args, **self._kwargs)
  File "problema3.py", line 13, in productor
    buffer[indice]=item
IndexError: list assignment index out of range

```

Se puede observar que el comportamiento es errático, y en algunos casos produce un error al querer acceder fuera del rango de la lista. Por ejemplo en la primera salida, en la cuarta línea el productor coloca 4 en la posición 2, pero también coloca el 4, en la misma posición, sin que se haya consumido. Entre otros comportamientos errados.

Solución 3 - Empleando Semaphore Objects

problema1.py	solucion1.py	problema2.py	solucion2.py	problema3.py	solucion3.py
<pre> 1 import threading as th 2 3 indice = -1 4 buffer = [None, None, None, None, None] 5 6 def productor(): 7 for n in range(20): 8 global indice 9 global buffer 10 item = n*n 11 libre.acquire() 12 mutex.acquire() 13 indice += 1 14 buffer[indice]=item 15 print(f"productor {indice} {item} {buffer}", flush=True) 16 mutex.release() 17 ocupado.release() 18 19 def consumidor(): 20 item = None 21 for _ in range(20): 22 global indice 23 global buffer 24 ocupado.acquire() 25 mutex.acquire() 26 item = buffer[indice] 27 buffer[indice]=None 28 print(f"consumidor {indice} {item} {buffer}", flush=True) 29 indice -= 1 30 mutex.release() 31 libre.release() 32 33 libre = th.Semaphore(value=5) 34 ocupado = th.Semaphore(value=0) 35 mutex = th.Semaphore(value=1) 36 37 t=th.Thread(target=productor) 38 t.start() 39 t=th.Thread(target=consumidor) 40 t.start() </pre>					

```

alejandrob@abdebian:Concurrencia$ python3 solucion3.py
productor 0 0 [0, None, None, None, None]
productor 1 1 [0, 1, None, None, None]
productor 2 4 [0, 1, 4, None, None]
productor 3 9 [0, 1, 4, 9, None]
productor 4 16 [0, 1, 4, 9, 16]
consumidor 4 16 [0, 1, 4, 9, None]
consumidor 3 9 [0, 1, 4, None, None]
consumidor 2 4 [0, 1, None, None, None]
consumidor 1 1 [0, None, None, None, None]
consumidor 0 0 [None, None, None, None, None]
productor 0 25 [25, None, None, None, None]
productor 1 36 [25, 36, None, None, None]
productor 2 49 [25, 36, 49, None, None]
productor 3 64 [25, 36, 49, 64, None]
productor 4 81 [25, 36, 49, 64, 81]
consumidor 4 81 [25, 36, 49, 64, None]
consumidor 3 64 [25, 36, 49, None, None]
consumidor 2 49 [25, 36, None, None, None]
consumidor 1 36 [25, None, None, None, None]
consumidor 0 25 [None, None, None, None, None]
productor 0 100 [100, None, None, None, None]
productor 1 121 [100, 121, None, None, None]
productor 2 144 [100, 121, 144, None, None]
productor 3 169 [100, 121, 144, 169, None]
productor 4 196 [100, 121, 144, 169, 196]
consumidor 4 196 [100, 121, 144, 169, None]
consumidor 3 169 [100, 121, 144, None, None]
consumidor 2 144 [100, 121, None, None, None]
consumidor 1 121 [100, None, None, None, None]
consumidor 0 100 [None, None, None, None, None]
productor 0 225 [225, None, None, None, None]
productor 1 256 [225, 256, None, None, None]
productor 2 289 [225, 256, 289, None, None]
productor 3 324 [225, 256, 289, 324, None]
productor 4 361 [225, 256, 289, 324, 361]
consumidor 4 361 [225, 256, 289, 324, None]
consumidor 3 324 [225, 256, 289, None, None]
consumidor 2 289 [225, 256, None, None, None]
consumidor 1 256 [225, None, None, None, None]
consumidor 0 225 [None, None, None, None, None]
alejandrob@abdebian:Concurrencia$ █

```

Problema 4

problema1.py	solucion1.py	problema2.py	solucion2.py	problema3.py	solucion3.py	problema4.py
<pre> 1 import threading as th 2 3 contador = [0,0] 4 5 def hilo(num): 6 for i in range(1,11): 7 t = num 8 print(f"Hilo {num} valor de {i}",flush=True) 9 other = contador[1-t] 10 contador[t] = other +1 11 12 ths = [] 13 for i in range(2): 14 t=th.Thread(target=hilo, args=(i,)) 15 ths.append(t) 16 17 for i in range(2): 18 ths[i].start() 19 20 for i in range(2): 21 ths[i].join() 22 23 print(f"contador de hilo 0 {contador[0]}") 24 print(f"contador de hilo 1 {contador[1]}") </pre>						

El objetivo en este programa es que cada hilo i , tuviera a *contador*[i] como su contador de vueltas, el que está marcado por el lazo for. Pero la cuenta se hace en función del contador del otro hilo y esto funciona sólo si la ejecución es estrictamente alternante. Es decir uno a uno. De lo contrario los resultados serán inesperados, observe abajo dos ejecuciones:

```
alejandro@abdebian:Concurrencia$ python3 problema4.py
Hilo 0 valor de 1
Hilo 0 valor de 2
Hilo 0 valor de 3
Hilo 0 valor de 4
Hilo 0 valor de 5
Hilo 0 valor de 6
Hilo 1 valor de 1
Hilo 1 valor de 2
Hilo 1 valor de 3
Hilo 1 valor de 4
Hilo 1 valor de 5
Hilo 0 valor de 7
Hilo 1 valor de 6
Hilo 1 valor de 7
Hilo 0 valor de 8
Hilo 1 valor de 8
Hilo 0 valor de 9
Hilo 1 valor de 9
Hilo 0 valor de 10
Hilo 1 valor de 10
contador de hilo 0 9
contador de hilo 1 10
```

```
alejandro@abdebian:Concurrencia$ python3 problema4.py
Hilo 0 valor de 1
Hilo 0 valor de 2
Hilo 0 valor de 3
Hilo 0 valor de 4
Hilo 0 valor de 5
Hilo 0 valor de 6
Hilo 0 valor de 7
Hilo 0 valor de 8
Hilo 1 valor de 1
Hilo 1 valor de 2
Hilo 1 valor de 3
Hilo 1 valor de 4
Hilo 1 valor de 5
Hilo 1 valor de 6
Hilo 1 valor de 7
Hilo 1 valor de 8
Hilo 1 valor de 9
Hilo 0 valor de 9
Hilo 0 valor de 10
Hilo 1 valor de 10
contador de hilo 0 5
contador de hilo 1 6
```

Solución 4 – Empleando Barrier Objects

problema1.py	solucion1.py	problema2.py	solucion2.py	problema3.py	solucion3.py	problema4.py	solucion4.py
--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------

```
1 import threading as th
2
3 contador = [0,0]
4
5 def hilo(num):
6     for i in range(1,11):
7         t = num
8         print(f"Hilo {num} valor de {i}",flush=True)
9         other = contador[1-t]
10        b1.wait()
11        contador[t] = other + 1
12        b2.wait()
13
14 b1 = th.Barrier(2)
15 b2 = th.Barrier(2)
16
17 ths = []
18 for i in range(2):
19     t=th.Thread(target=hilo, args=(i,))
20     ths.append(t)
21
22 for i in range(2):
23     ths[i].start()
24
25 for i in range(2):
26     ths[i].join()
27
28 print(f"contador de hilo 0 {contador[0]}")
29 print(f"contador de hilo 1 {contador[1]}")
```

```
alejandro@abdebian:Concurrencia$ python3 solucion4.py
Hilo 0 valor de 1
Hilo 1 valor de 1
Hilo 0 valor de 2
Hilo 1 valor de 2
Hilo 0 valor de 3
Hilo 1 valor de 3
Hilo 0 valor de 4
Hilo 1 valor de 4
Hilo 0 valor de 5
Hilo 1 valor de 5
Hilo 1 valor de 6
Hilo 0 valor de 6
Hilo 1 valor de 7
Hilo 0 valor de 7
Hilo 1 valor de 8
Hilo 0 valor de 8
Hilo 1 valor de 9
Hilo 0 valor de 9
Hilo 1 valor de 10
Hilo 0 valor de 10
contador de hilo 0 10
contador de hilo 1 10
```

Problema 5

Se tiene el siguiente programa en Python (*alterna.py*)

```
alterna.py x
1 import threading as th
2 import random as rand
3 import time
4
5 def izq_der():
6     r = rand.randint(10,20)
7     time.sleep(r/100)
8     print("----->", flush=True)
9
10 def der_izq():
11     r = rand.randint(10,20)
12     time.sleep(r/100)
13     print("<-----", flush=True)
14
15 ths = []
16 for i in range(10):
17     t = th.Thread(target=izq_der)
18     ths.append(t)
19
20 for i in range(10):
21     t = th.Thread(target=der_izq)
22     ths.append(t)
23
24 for i in range(20):
25     ths[i].start()
26
27 for i in range(20):
28     ths[i].join()
29
```

cuya salida, al ejecutarse, es la siguiente (usted puede obtener una salida diferente):

```
alejandro@abdebian:2020-1$ python3 alterna.py
----->
----->
<-----
<-----
<-----
----->
<-----
----->
----->
----->
----->
----->
<-----
<-----
<-----
----->
<-----
----->
<-----
----->
<-----
alejandro@abdebian:2020-1$
```

- Sincronizar los hilos de forma que haya alternancia estricta entre el hilo que invoca `izq_der` y el otro hilo que invoca `der_izq`.
- En general, al código base le puede añadir, pero no le puede eliminar nada. Solo puede añadir las líneas necesarias a las funciones `izq_der` y `der_izq`, para lograr la sincronización. También puede añadir variables globales.

Solución 5 Empleando Condition Objects

```
1 import threading as th
2 import random as rand
3 import time
4
5 c = th.Condition()
6 turno = "izq"
7
8 def izq_der():
9     global turno
10    c.acquire()
11    while turno != "izq":
12        c.wait()
13    r = rand.randint(10,20)
14    time.sleep(r/100)
15    print("----->", flush=True)
16    turno = "der"
17    c.notifyAll()
18    c.release()
19
20 def der_izq():
21     global turno
22     c.acquire()
23     while turno != "der":
24         c.wait()
25     r = rand.randint(10,20)
26     time.sleep(r/100)
27     print("<-----", flush=True)
28     turno = "izq"
29     c.notifyAll()
30     c.release()
31
32 ths = []
33 for i in range(10):
34     t = th.Thread(target=izq_der)
35     ths.append(t)
36
37 for i in range(10):
38     t = th.Thread(target=der_izq)
39     ths.append(t)
40
41 for i in range(20):
42     ths[i].start()
43
44 for i in range(20):
45     ths[i].join()
```


Timer Objects

Esta clase representa una acción que debería ser ejecutada sólo después que una cierta cantidad de tiempo ha transcurrido (como un temporizador). Timer es una subclase de Thread y como tal también funciona como un ejemplo de creación de hilos.

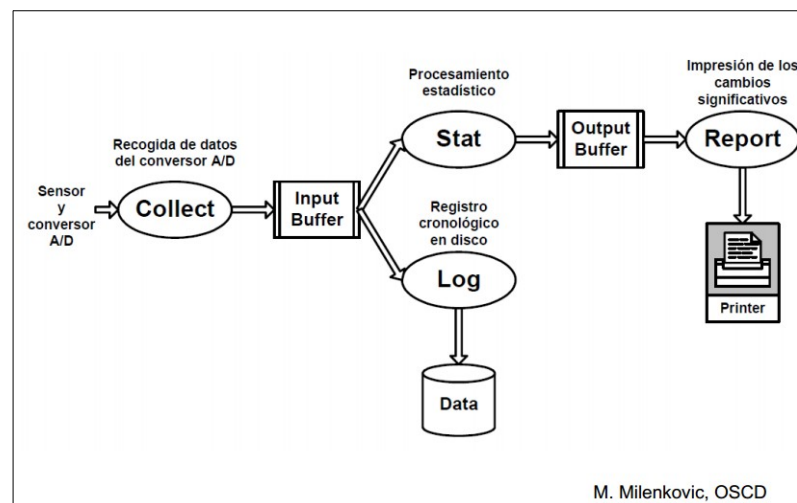
Los Timer son iniciados, al igual que los Thredas, invocando el método `start()`. El temporizador puede ser detenido (antes que su acción haya empezado) invocando el método `cancel()`.

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start() # after 30 seconds, "hello, world" will be printed
```

Ejercicios

1.- Se desea simular en Python la sincronización de hilos en le caso del sistema de adquisición de datos, propuesto por Millan Milenkovic, compuesto por los siguientes hilos: Collect, Log, Stat y Report. El hilo Collect adquiere los datos del DAC y los deposita en un *input buffer* (vea dibujo) de entrada, el cual es leído por los procesos Stat y Log. Collect no puede depositar los datos hasta que hayan acabado de procesar Stat y Log. A su vez Stat y Log no pueden empezar a leer el *input buffer* hasta que Collect haya llenado el *buffer*. Stat realiza cálculos y deposita sus resultados en *output buffer* (vea dibujo), el cual es leído por el proceso Report. Report no puede empezar hasta que Stat haya terminado de llenar el *buffer*. A su vez Stat no puede empezar si Report no ha terminado.



A continuación el programa en Python carente de sincronización:

```
import threading

def Collect():
    while True:
        print('C',end=' ')

def Log():
    while True:
        print('L',end=' ')

def Stat():
    while True:
        print('S',end=' ')

def Report():
    while True:
        print('R',end=' ')

if __name__ == "__main__":

    h1=threading.Thread(target=Collect)
    h2=threading.Thread(target=Log)
    h3=threading.Thread(target=Stat)
    h4=threading.Thread(target=Report)
    h1.start()
    h2.start()
    h3.start()
    h4.start()
```

Empleando barreras en Python, sincronice los hilos según el enunciado.

La salida debe ser análoga a la siguiente:

```
alejandro@abdebian:S0$ python3 dac.py
```

2.- Dado el siguiente programa

```
agua.py
1  import threading as th
2
3
4  def Hidrogeno():
5      while True:
6          print("H",end="")
7
8  def Oxigeno():
9      while True:
10         print("O",end="")
11
12 def Cambiodelinea():
13     while True:
14         print()
15
16 t = th.Thread(target=Hidrogeno)
17 t.start()
18 t = th.Thread(target=Oxigeno)
19 t.start()
20 t = th.Thread(target=Cambiodelinea)
21 t.start()
```

como es de esperarse, imprime de forma desordenada los caracteres: 'O', 'H' y '\n'. Se le pide sincronizar usando *cualquier herramienta de concurrencia de Python* para que siempre se imprima 2 letras 'H', una letra 'O' y un cambio de línea. Las siguientes secuencias son válidas:

HHO\n
HOH\n
OHH\n

3.- Se tiene un programa en Python que tiene 5 hilos. Cada hilo imprime uno de los siguientes caracteres: A,B, C, D, E. Usted debe sincronizarlos para que cumplan las siguientes exigencias. Elabore un programa por cada caso.

- a) La secuencia permitida es: ABCDEABCDEABCDEABCDEABCDE ...
- b) La secuencia permitida es: ACDEBCDEACDEBCDEACDEBCDEACDEBCDE...
- c) La secuencia permitida es: (A ó B)CDE(A ó B)CDE(A ó B)CDE(A ó B)CDE ...
- d) La secuencia permitida es: (A ó B)CE(A ó B)(A ó B)DE(A ó B)CE(A ó B)(A ó B)DE ...

Prof. Alejandro T. Bello Ruiz