

Arquitectura y Sistemas Operativos

Tecnicatura Universitaria en Programación (TUP) – UTN FRBB

Profesor: Gustavo Ramoscelli

Ayudantes: Leandro M. Regolf - Sergio Antozzi

Trabajo Práctico Nº 3

Procesos multihilados (multi-threads)

- 1- Objetivo y generalidades
- 2- Hilos
- 3- Condición de carrera (Race Condition)
- 4- Detección y corrección del problema
- 5- Entrega de TP3

Antes de empezar

Se recomienda recorrer y dar una leída completa a todo el enunciado del trabajo práctico para tener una idea general y recién luego iniciar su resolución, para evitar trabajar de mas, confusiones, etc.

1- Objetivo y generalidades

El objetivo de esta parte del práctico es redondear el tema y ver algunas de las **ventajas y dificultades** a las que nos enfrentamos al programar procesos **multihilados (o multi-threads)**.

Esta vez utilizando el lenguaje de programación **Python**.

La programación multihilo nos permite, dentro del desarrollo de una aplicación, definir diversas tareas para ser ejecutadas o bien de una vez, o bien en paralelo o “al mismo tiempo”, teniendo en cuenta que todas estas tareas forman parte de el **mismo proceso**.

El **Sistema Operativo** administrará un solo proceso con diversas tareas a ejecutar. El **planificador** del sistema operativo, o del intérprete del lenguaje en este caso, se encargará de que cuando le toque ejecutarse a nuestro proceso, las tareas del mismo se vayan alternando (en **hilos**), con la diferencia que, al ser de un mismo proceso, no se liberará espacio de memoria ni procesador, como si se haría en el caso de alternar procesos.

Esto agiliza por lo tanto la ejecución de nuestro proceso que además puede simular **paralelizar** la ejecución de varias tareas a la vez.

Generalmente sucede que más de un hilo debe acceder a un **recurso compartido**, generando **carreras críticas**, por lo que es necesario implementar algún mecanismo de control de acceso para evitar fallas o funcionamientos inesperados.

2- Hilos

Para realizar programación multihilo, debemos indicar, en nuestro código de programación, utilizando los comandos o instrucciones correspondientes a cada lenguaje, cuáles tareas deseamos ejecutar en **diferentes hilos**.

Dado el siguiente programa (**tareas_SIN_hilos.py**) en Python:

```
1  import time
2
3
4  def tarea_1():
5      momento_arranque = time.perf_counter()
6      print('Inicio tarea 1')
7
8      for x in range(100000000):
9          pass
10
11     print('Fin tarea 1')
12     momento_parada = time.perf_counter()
13
14     print(f'Tomó {momento_parada - momento_arranque: 0.5f} segundos completar la tarea 1')
15
16
17  def tarea_2():
18     momento_arranque = time.perf_counter()
19     print('Inicio tarea 2')
20     time.sleep(1)
21     print('Fin tarea 2')
22     momento_parada = time.perf_counter()
23
24     print(f'Tomó {momento_parada - momento_arranque: 0.5f} segundos completar la tarea 2')
25
26
27  def tarea_3():
28     momento_arranque = time.perf_counter()
29     print('Inicio tarea 3')
30     time.sleep(4)
31     print('Fin tarea 3')
32     momento_parada = time.perf_counter()
33
34     print(f'Tomó {momento_parada - momento_arranque: 0.5f} segundos completar la tarea 3')
35
36
37  momento_arranque = time.perf_counter()
38  tarea_1()
39  tarea_2()
40  tarea_3()
41  momento_parada = time.perf_counter()
42
43
44  print(f'Tomó un total de {momento_parada - momento_arranque: 0.5f} segundos completar todas las tareas')
45  |
```

En la línea 1 importamos el módulo **time**, ya incluido en python (los módulos son similares a las librerías de C)

Vemos que consta de 3 funciones (y un cuerpo principal que invoca a cada una de ellas):

tarea_1(), **tarea_2()** y **tarea_3()**

tarea_1() se ejecuta a la máxima velocidad posible, pero su tiempo de ejecución es **variable**, ya que depende de la velocidad real de la máquina donde se está ejecutando, la carga de CPU al momento de ejecución, etc. Podemos decir que la velocidad de ejecución **depende** principalmente de la velocidad de procesamiento de la máquina en donde se ejecuta.

tarea_2() tiene una demora **fija** (arbitraria) de 1 seg, digamos que la espera **no depende** la velocidad de procesamiento de la máquina

tarea_3() tiene una demora **fija** de 4 seg, digamos que la espera **no depende** la velocidad de procesamiento de la máquina

Las líneas 37 y 41 nos permiten tomar los tiempos de arranque y parada para luego poder calcular el tiempo que se demoró la ejecución del programa en su totalidad.

Ejecutar **varias veces** el código

- ¿Qué se puede notar con respecto al tiempo de ejecución? ¿Es predecible?
- Nombrar un proceso o función de la vida real que pueden ser considerados procesos de “máxima velocidad posible” que dependen casi exclusivamente de la velocidad de la máquina que los ejecuta (ej. Ordenar una lista)
- Nombrar un proceso o función de la vida real que pueden ser considerados procesos de “velocidad de respuesta no dependiente de la velocidad de procesamiento” o que sea de naturaleza impredecible o externa (ej. Leer un archivo externo)

Modifiquemos y mejoremos el código y generemos **tareas_CON_hilos.py**

```
1  import time
2  import threading
3
4
5  def tarea_1():
6      momento_arranque = time.perf_counter()
7      print('Inicio tarea 1')
8
9      for x in range(100000000):
10         pass
11
12     print('Fin tarea 1')
13     momento_parada = time.perf_counter()
14
15     print(f'Tomó {momento_parada - momento_arranque: 0.5f} segundos completar la tarea 1')
16
17
18  def tarea_2():
19     momento_arranque = time.perf_counter()
20     print('Inicio tarea 2')
21     time.sleep(1)
22     print('Fin tarea 2')
23     momento_parada = time.perf_counter()
24
25     print(f'Tomó {momento_parada - momento_arranque: 0.5f} segundos completar la tarea 2')
26
27
28  def tarea_3():
29     momento_arranque = time.perf_counter()
30     print('Inicio tarea 3')
31     time.sleep(4)
32     print('Fin tarea 3')
33     momento_parada = time.perf_counter()
34
35     print(f'Tomó {momento_parada - momento_arranque: 0.5f} segundos completar la tarea 3')
36
37
38  momento_arranque = time.perf_counter()
39  hilo1 = threading.Thread(target=tarea_1)
40  hilo2 = threading.Thread(target=tarea_2)
41  hilo3 = threading.Thread(target=tarea_3)
42
43  hilo1.start()
44  hilo2.start()
45  hilo3.start()
46
47  hilo1.join()
48  hilo2.join()
49  hilo3.join()
50  momento_parada = time.perf_counter()
51
52
53  print(f'Tomó un total de {momento_parada - momento_arranque: 0.5f} segundos completar todas las tareas')
54  |
```

En la línea 2 importamos el módulo **threading**, que nos permitirá crear y administrar hilos en Python

Vemos que las funciones **no se modifican**

Lo que sí cambia la manera invocar las funciones, ya que las declaramos como **hilos independientes**

39 a 41 creamos cada hilo asignándole una función a cada uno con **target**

43 a 45 indicamos que comiencen a correr dichos hilos con **start**

47 a 49 esperamos su culminación con **join**

- Ejecutar **varias veces** el código
- ¿Qué se puede notar con respecto al tiempo de ejecución? ¿Se mejoró el tiempo de respuesta con respecto al mismo programa sin hilos?
- ¿Completan las funciones su ejecución en el orden establecido?
- Nombrar un escenario real donde el multi-hilado puede mejorar considerablemente el tiempo de respuesta de un sistema (ej. Carga de una página WEB en un navegador)

3- Condición de carrera (Race Condition)

Dado el siguiente programa (**sumador-restador.py**) en Python:

```
1  import time
2
3
4  acumulador = 0
5
6
7  def sumador():
8      global acumulador
9
10     for _ in range(1000):
11         tmp = acumulador
12
13         time.sleep(0)
14         tmp = tmp + 5
15         time.sleep(0)
16
17         acumulador = tmp
18
19
20  def restador():
21      global acumulador
22
23     for _ in range(1000):
24         tmp = acumulador
25
26         time.sleep(0)
27         tmp = tmp - 5
28         time.sleep(0)
29
30         acumulador = tmp
31
32
33  sumador()
34  restador()
35
36
37  print(f'El valor calculado final es: {acumulador}')
38
```

Notar que la variable `acumulador` es una variable **global**, que es utilizada tanto dentro como fuera de las funciones, siendo esta siempre la misma variable y conservando su valor en **todos los ámbitos**

- Ejecutar varias veces el código

Podemos ver que dicho código utilizará una variable `acumulador`, la función **sumador()** incrementará dicha variable en 5 unidades un mil veces, y la función **restador()** decrementará en 5 unidades la misma variable la misma cantidad de veces, lo que da un resultado neto igual a cero al final de la ejecución del programa.

Modifiquemos el código utilizando hilos de ejecución y generemos **sumador-restador_CON_race.py**

```
conrace.py X
conrace.py > [e] momento_parada
1  import time
2  import threading
3
4
5  acumulador = 0
6
7  def sumador():
8      global acumulador
9
10     for x in range(1000000):
11         acumulador = acumulador + 5
12
13  def restador():
14      global acumulador
15
16     for x in range(1000000):
17         acumulador = acumulador - 5
18
19
20  momento_arranque = time.perf_counter()
21  thr1 = threading.Thread(target=sumador)
22  thr2 = threading.Thread(target=restador)
23
24  thr1.start()
25  thr2.start()
26
27  thr1.join()
28  thr2.join()
29  momento_parada = time.perf_counter()
30
31
32  print(f'El valor final es: {acumulador}')
33  print(f'Tomó un total de {momento_parada - momento_arranque: 0.5f} segundos completar las tareas.')
34
```

- Ejecutar **varias veces** el código
- ¿Qué se puede notar con respecto al tiempo de ejecución?
- ¿Qué sucede con el **valor final del acumulador**?
- ¿Por qué sucede esto?
TIP: cambios de contexto en medio de acceso a zona crítica y actualización de valor de variable
- ¿Cómo se puede corregir esta condición de carrera sin dejar de utilizar hilos?

4- Detección y corrección del problema

Modifiquemos el código y generemos `sumador-restador_SIN_race.py`

```
1  import time
2  import threading
3
4
5  acumulador = 0
6
7
8  def sumador(proteccion):
9      global acumulador
10
11      for _ in range(1000):
12          with proteccion:
13              tmp = acumulador
14
15              time.sleep(0)
16              tmp = tmp + 5
17              time.sleep(0)
18
19              acumulador = tmp
20
21
22  def restador(proteccion):
23      global acumulador
24
25      for _ in range(1000):
26          with proteccion:
27              tmp = acumulador
28
29              time.sleep(0)
30              tmp = tmp - 5
31              time.sleep(0)
32
33              acumulador = tmp
34
35
36  proteccion = threading.Lock()
37
38
39  hilo1 = threading.Thread(target=sumador, args=(proteccion,))
40  hilo2 = threading.Thread(target=restador, args=(proteccion,))
41
42  hilo1.start()
43  hilo2.start()
44
45  hilo1.join()
46  hilo2.join()
47
48
49  print(f'El valor calculado final es: {acumulador}')
50
```

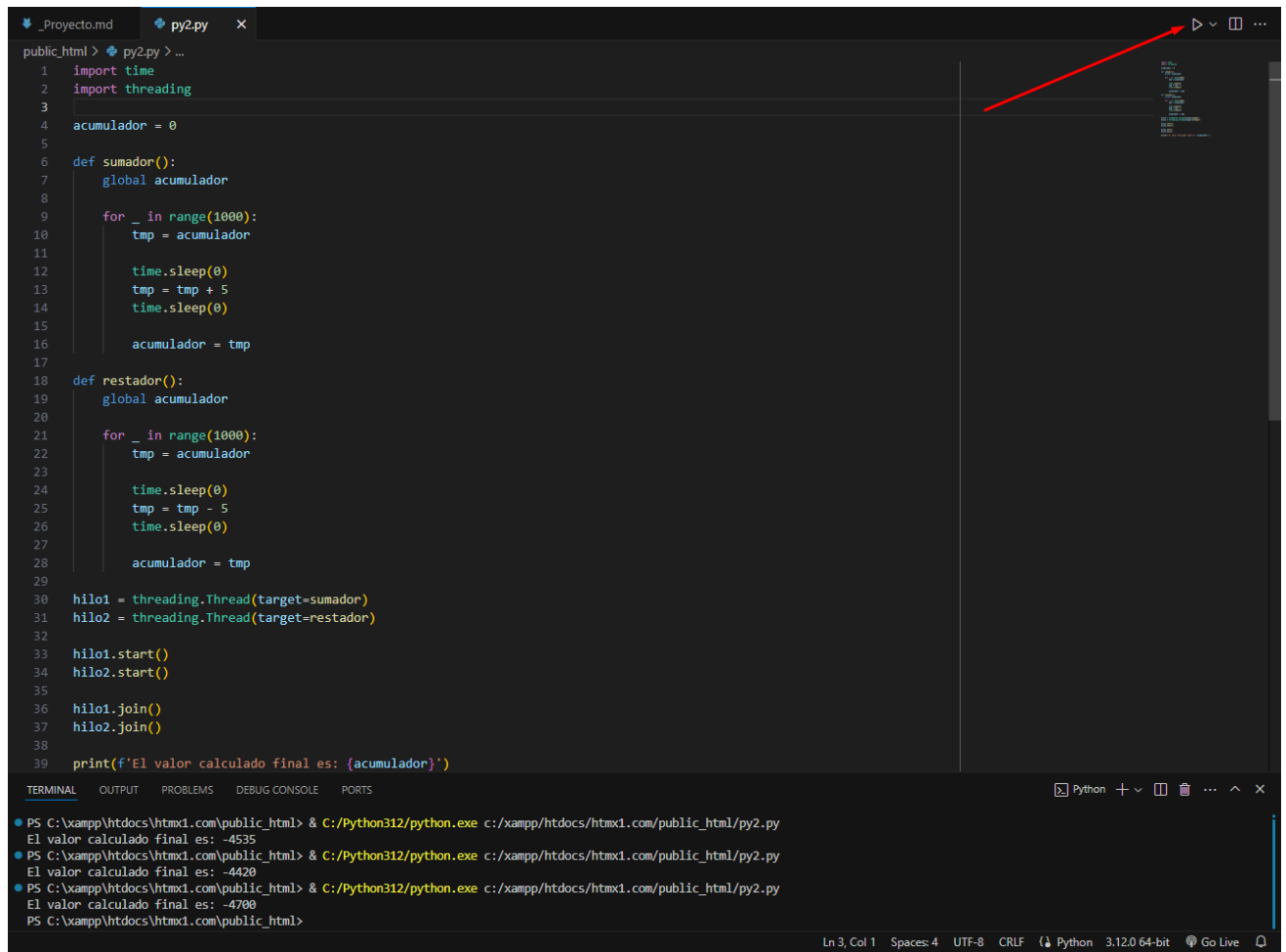
Creamos una variable **protección** (no global) para ser utilizada como “bloqueador de hilo”, esta funcionalidad es **provista** por el módulo **threading**

Cambia la manera de ejecutar las funciones, al declararlas como **hilos independientes** debemos indicarles también que deben utilizar el **candado** o **protección de hilo** con la variable **protección**

Cambian también las funciones, ya que ahora deben utilizar el **candado** con la variable **protección**. Entonces, antes de acceder a la zona crítica se debe “tomar posesión” del candado (**acquire**), de esta manera se impide que ningún otro hilo pueda acceder a ese recurso. Esto se realiza con la instrucción **with** dentro de las funciones **sumador** y **restador**, justo antes de acceder al bloque de código que

modifica el valor de la variable compartida.

Luego se debe “dejar la posesión” del candado (**release**) para permitir el acceso al recurso compartido a los hilos que lo requieran.



```
public_html > py2.py > ...
1 import time
2 import threading
3
4 acumulador = 0
5
6 def sumador():
7     global acumulador
8
9     for _ in range(1000):
10         tmp = acumulador
11
12         time.sleep(0)
13         tmp = tmp + 5
14         time.sleep(0)
15
16         acumulador = tmp
17
18 def restador():
19     global acumulador
20
21     for _ in range(1000):
22         tmp = acumulador
23
24         time.sleep(0)
25         tmp = tmp - 5
26         time.sleep(0)
27
28         acumulador = tmp
29
30 hilo1 = threading.Thread(target=sumador)
31 hilo2 = threading.Thread(target=restador)
32
33 hilo1.start()
34 hilo2.start()
35
36 hilo1.join()
37 hilo2.join()
38
39 print(f'El valor calculado final es: {acumulador}')
```

TERMINAL OUTPUT:

```
PS C:\xampp\htdocs\html1.com\public_html> & C:/Python312/python.exe c:/xampp\htdocs/html1.com/public_html/py2.py
El valor calculado final es: -4535
PS C:\xampp\htdocs\html1.com\public_html> & C:/Python312/python.exe c:/xampp\htdocs/html1.com/public_html/py2.py
El valor calculado final es: -4420
PS C:\xampp\htdocs\html1.com\public_html> & C:/Python312/python.exe c:/xampp\htdocs/html1.com/public_html/py2.py
El valor calculado final es: -4700
PS C:\xampp\htdocs\html1.com\public_html>
```

- Ejecutar **varias veces** el código
- ¿Qué sucede con el valor final del acumulador?
- ¿Qué se puede notar con respecto al tiempo de ejecución?

5- Entrega de TP3

Para la entrega del TP3 se procederá de manera similar a la entrega del TP2:

Se creará en nuestro repositorio local, una carpeta TP3, donde guardarán algunas capturas de pantalla del código corriendo en VSCode (varias ejecuciones del código) donde se puedan apreciar algunas de las problemáticas presentadas,

como así también un **archivo de texto** con las **respuestas** a las preguntas formuladas.

Luego se deberá pushear dicha carpeta a nuestro propio repositorio remoto, de la manera practicada anteriormente:

- git add .
- git commit -m "entrega TP3"
- git push origin main