Universidad Nacional Experimental Simón Bolívar
Lenguajes de Programación I
Prof. Ricardo Monascal

Caracas, 12 de Abril 2021

Alumno y Carné:
Victoria Torres 12-11468

Examen 3: Pregunta 2

2. Escoja algún lenguaje de programación de alto nivel y de propósito general cuyo nombre tenga la misma cantidad de caracteres que su primer nombre.

Utilizando mi apellido **Torres**, seleccione el lenguaje de programación **Python** de entre los que poseen 6 caracteres en su nombre.

(a) De una breve descripción de los mecanismos de concurrencia disponibles en su lenguaje.

Python incluye herramientas algo sofisticadas para manejar operaciones concurrentes utilizando proceso o hilos que permiten que hasta programas considerablemente simples se ejecuten más rápido aplicando técnicas para que partes del trabajo se ejecuten al mismo tiempo haciendo uso de estos módulos. A su vez, estos permiten que el programador también pueda implementar co-rutinas y tareas asíncronas.

Los nombres de dichos módulos en cuestión son:

- subprocess
- signal
- threading
- multiprocessing
- asyncio
- concurrent.futures

i. Diga si su lenguaje provee capacidades nativas para concurrencia, usa librerías o depende de herramientas externas.

Python para manejar concurrencia depende de módulos. Estos proporcionan todas las herramientas que necesitamos para manejar concurrencia, crear co-rutinas, etc, por lo cual tendremos que importarlos para poder trabajar con ellos.

Veamos una breve descripción de lo que nos ofrecen los antes mencionados:

1. **subprocess:** Proporciona una interfaz para que creemos y nos comuniquemos con procesos secundarios. Es especialmente bueno para ejecutar programas que producen o consumen texto, puesto que la interfaz admite el paso de datos de ida y vuelta a través de los canales de entrada y salida estándar del nuevo proceso.

2. **signal:** Expone el mecanismo de señal de Unix para enviar eventos a otros procesos. Estas señales se procesan de manera asíncrona generalmente interrumpiendo lo que hace el programa en el momento en que llega la señal. Normalmente la señalización es útil como un sistema de mensajes, pero otras técnicas de comunicación entre procesos son mas confiables y pueden entregar mensajes mas complejos.

3. **threading:** Es uno de los más utilizados y conocidos. Este incluye una interfaz de alto nivel orientada a objetos para trabajar con concurrencia desde Python permitiéndonos usar la clase Thread para crear hilos y manipularlos con los métodos que pone a nuestra disposición. Los objetos Thread se ejecutan al mismo tiempo dentro del mismo proceso y comparten memoria. Hacer uso de hilos es una forma sencilla de escalar para tareas que son mas enlazadas a la E/S que al CPU.

4. **multiprocessing:** Este es reflejo del modulo threading, pero en vez de proveer una clase Thread, proporciona una clase Process de quien crearemos procesos que manipularemos a su vez con los métodos que esta pone a nuestra disposición. Cada objeto Process corresponde a un verdadero proceso del sistema sin memoria compartida, pero además este modulo proporciona funciones para compartir datos y pasar mensajes entre ellos de manera que, en muchos casos, la conversión de hilos a procesos es tan simple como cambiar unas pocas declaraciones *import*.

5. **asyncio:** Proporciona un marco de concurrencia y manejo asíncrono de entrada/salida usando un sistema de protocolo basado en clases o co-rutinas. De hecho al crear este modulo fueron reemplazados los módulos antiguos *asyncore* y *asynchat* que, aún se encuentran disponibles pero, son considerados obsoletos.

6. **concurrent.futures:** Proporciona la implementación de ejecutores basados en hilos y en procesos para administrar grupos de recursos para ejecutar tareas concurrentes.

ii. Explique la creación/manejo de tareas concurrentes, así como el control de la memoria compartida y/o pasaje de mensajes.

Para este apartado voy a basar la explicación en el modulo *threading* que es uno de los mas utilizados, ya que cada uno de los módulos tiene sus propias instrucciones y mañas asociadas.

Sabemos que le propósito de threading es permitirnos gestionar varios hilos de ejecución para gestionar operaciones concurrentes dentro de un proceso, es decir, podremos ejecutar múltiples operaciones de forma simultanea en el mismo espacio de proceso.

La forma mas sencilla de comenzar a usar Thread es simplemente crear un hilo, es decir, una instancia con una función de destino y luego llamar a *start()* para que comience a funcionar.
Veamos un ejemplo de esto:

```
import threading

#Funcion que ejecutaran los hijos
def runner
    print(“runner”)

#Un arreglito para colocar los hilos
threads = []

#Con el ciclo crearemos 5 hilos a quienes les indicamos que su trabajo es ejecutar runner, los ingresamos
#al arreglo de hilos y luego los iniciamos con start()
for  i in range(5):
   t = threading.Thread(target=runner)
```

```
    threads.append(t)
    t.start()
```

Si lo ejecutamos este ejemplo nos imprimirá:

```
runner
runner
runner
runner
runner
```

También podemos pasarle argumentos a los hilos para indicarle también que trabajo hacer, en este caso cualquier tipo de objeto puede ser pasado al hilo. Por ejemplo:

```
import threading

#Funcion que ejecutaran los hijos
def runner(num)
     print("runner nro %s" % num)

#Un arreglito para colocar los hilos
threads = []

#Misma idea que antes pero ahora le pasamos un numero como argumento al hilo, que luego imprimirá
for  i in range(5):
    t = threading.Thread(target=runner, args=(i,))
    threads.append(t)
    t.start()
```

Si lo ejecutamos este ejemplo nos imprimirá:

```
runner nro 0
runner nro 1
runner nro 2
runner nro 3
runner nro 4
```

Con estos ejemplos Podemos ver como entonces podemos ir creando hilos y les asignamos la tarea que van a realizar a cada uno luego a medida que los vamos inciando con *start()* estos comienzan a realizar su labor de manera concurrente.

Sin embargo, aunque le punto de usar múltiples hilos es ejecutar separadamente operaciones al mismo tiempo, hay momentos en los que es importante ser capaz de sincronizar las operaciones en dos o mas hilos.

Una forma de hacer esto, de forma sencilla y segura, son los objetos Event. Estos lo que hacen es gestionar una *bandera* interna que los llamadores pueden controlar con los métodos *set()* y *clear()*. Otros hilos también pueden usar *wait()* para pausar hasta que se establezca la *bandera*, bloqueando efectivamente el avance hasta que se permita continuar.

Veamos un ejemplo un poco más complejo "copy-pasteado" de internet:

```
import logging
import threading
import time


def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    logging.debug('wait_for_event starting')
    event_is_set = e.wait()
    logging.debug('event set: %s', event_is_set)


def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    while not e.is_set():
        logging.debug('wait_for_event_timeout starting')
        event_is_set = e.wait(t)
        logging.debug('event set: %s', event_is_set)
        if event_is_set:
            logging.debug('processing event')
        else:
            logging.debug('doing other work')


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

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

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

logging.debug('Waiting before calling Event.set()')
time.sleep(0.3)
e.set()
logging.debug('Event is set')
```

En este el método *wait_for_event_timeout()* toma un argumento que representa el numero de segundos que el evento espera antes de que se agote el tiempo de espera y devuelve un booleano indicando si el evento esta configurado o no para que el que llama sepa por qué *wait()* retornó.

Por otro lado el método *is_set()* puede ser usado por separado en el evento sin temor a bloquear. Como tal en el ejemplo *wait_for_event_timeout()* comprueba el estatus del evento sin bloqueo indefinido, por otro lado el metodo *wait_for_event()* bloquea en la llamada a *wait()*, que no retorna hasta que el estado del evento cambie.

Haciendo cosas como estas podemos controlar la concurrencia de las tareas, de modo que cada hilo puede estar trabajando simultáneamente, pero si alguno debe esperar que otro termine para poder acceder a cierta información, o realizar algún cambio, o porque de repente necesiten pasarse un mensaje entre ellos, entonces funciones como estas nos ayudan a manejar la espera.

También es importante controlar el acceso a la memoria y recursos compartidos para prevenir corrupción o perdida de datos. Aquí, estructuras incorporadas en Python como Listas, Diccionarios, entre otros, son seguras para subprocesos como un efecto secundario de tener códigos de bytes atómicos para manipularlos ya que el bloqueo global del interprete se usa para proteger las estructuras de datos internas de Python. Otras estructuras de datos implementadas en Python, o los tipos más simples (como números enteros, flotantes, etc) no tienen esa protección, entonces para protegerse contra el acceso simultaneo a un objeto utilizan un objeto *Lock.*

Estos objetos Lock están en uno de dos estados *cerrado* o *abierto* (locked/unlocked). Estos se crean en estado abierto y tienen dos metodos básicos *acquire()* (adquirir) y *reléase()* (liberar). Cuando están es estado *abierto*, *acquire()* cambia al estado cerrado y retorna inmediatamente. Cuando el estado es *cerrado*, *acquire()* bloquea hasta que una llamada a *release()* en otro hilo lo cambie a abierto, luego la llamada a *acquire()* lo reestablecerá a cerrada y retorna. Cabe destacar que el metodo *reléase()* solo debe ser llamado en el estado cerrado para cambiar al estado abierto y retornar inmediatamente. Si se llegara a intentar liberar un objeto lock abierto, se lanzara un *RunTimeError*.

Si vemos este otro ejemplo “copy-pasteado”:

```
import threading

total = 0

def acumula5():
    global total
    contador = 0
    hilo_actual = threading.current_thread().getName()
    while contador < 20:
        print('Esperando para bloquear', hilo_actual)
        bloquea.acquire()
        try:
            contador = contador + 1
            total = total + 5
            print('Bloqueado por', hilo_actual, contador)
            print('Total', total)

        finally:
            print('Liberado bloqueo por', hilo_actual)
            bloquea.release()

bloquea = threading.Lock()
hilo1 = threading.Thread(name='h1', target=acumula5)
hilo2 = threading.Thread(name='h2', target=acumula5)
hilo1.start()
```

```
hilo2.start()
```

Aquí, estamos iniciando dos hilos que actualizan la variable global **total** donde se van acumulando números que son múltiplos de 5. Antes y después de cada actualización se produce un bloqueo y un desbloqueo respectivamente para justamente evitar que ambos hilos vayan a actualizar la variable al mismo tiempo.

Por otro lado, para conocer si otro hilo ha adquirido el bloqueo sin mantener al resto de subprocesos detenidos, hay que asignar al argumento *blocking* de *acquire()* el valor *False*, de esta manera se pueden realizar otros trabajos mientras se espera tener éxito en un bloqueo y controlar el número de reintentos realizados. También se puede utilizar el método *locked()* para verificar si un bloqueo se mantiene en un momento dado como sigue:

```
import threading

def acumula5():
    global total
    contador = 0
    hilo_actual = threading.current_thread().getName()
    num_intentos = 0
    while contador < 20:
        lo_consegui = bloquea.acquire(blocking=False)
        try:
            if lo_consegui:
                contador = contador + 1
                total = total + 5
                print('Bloqueado por', hilo_actual, contador)
                print('Total', total)
            else:
                num_intentos+=1
                print('Número de intentos de bloqueo',
                      num_intentos,
                      'hilo',
                      hilo_actual,
                      bloquea.locked())
                print('Hacer otro trabajo')

        finally:
            if lo_consegui:
                print('Liberado bloqueo por', hilo_actual)
                bloquea.release()

total = 0
bloquea = threading.Lock()
hilo1 = threading.Thread(name='h1', target=acumula5)
hilo2 = threading.Thread(name='h2', target=acumula5)
hilo1.start()
hilo2.start()
```

A veces tambien es útil permitir que mas de un hilo acceda a un recurso a la vez, mientras que todavía se limita el numero total, para esto podemos usar un *Semaphore*. Los *semáforos* son un instrumento de bloqueo más avanzado que usa un contador interno para controlar el número de hilos que pueden acceder de forma concurrente a una parte del código. Si el número de hilos que intentan

acceder supera, en un momento dado, al valor establecido, entonces se producirá un bloqueo que será liberado en la medida que los hilos no bloqueados vayan completando las operaciones previstas.

Como tal los semáforos se utilizan para restringir el acceso a recursos con capacidad limitada. A continuación, podemos ver un ejemplo:

```
import threading
import time

def descargando(semaforo):
    global activas
    nombre = threading.current_thread().getName()
    print('Esperando para descargar:', nombre)
    with semaforo:
        activas.append(nombre)
        print('Descargas activas', activas)
        print('...Descargando...', nombre)
        time.sleep(0.1)
        activas.remove(nombre)
        print('Descarga finalizada', nombre)

NUM_DESCARGAS_SIM = 3
activas = []
semaforo = threading.Semaphore(NUM_DESCARGAS_SIM)
for indice in range(1,6):
    hilo = threading.Thread(target=descargando,
                            name='D' + str(indice),
                            args=(semaforo,),)
    hilo.start()
```

Aquí se generan 5 hilos para simular una descarga simultanea de archivos. Las descargas podrán ser concurrentes hasta un máximo de 3 (que es el valor que se le coloco a la constante NUM_DESCARGAS_SIM, utilizada para declarar el objeto *Semaphore*).

iii. Describa el mecanismo de sincronización que utiliza el lenguaje.

Las primitivas de sincronización en Python mas populares son las definidas en el modulo estándar threading.py. La mayoría de los metodos de bloqueo (es decir, que bloquean ejecutar un hilo en particular hasta que se cumpla una condicion) de estas primitivas proporcionan la función de tiempo de espera opcional. Algunas de estas las hemos mencionado en la pregunta anterior.

La primitiva de sincronización mas simple en Python son los **bloqueos**, es decir, los objetos **Lock** que mencionamos anteriormente, quienes tienen dos estados *bloqueado* y *desbloqueado* y con los que podemos controlar el acceso a los recursos.

También podemos sincronizar los hilos con objetos **Event**, quienes también mencionamos anteriormente. A veces es necesario que varios hilos se puedan comunicar entre si para sincronizar sus trabajos y hasta para pasar mensajes. Con los objetos Event, por su capacidad de anunciar anuno o más

hilos (que esperan) que se ha producido un suceso para que puedan proseguir su ejecución. Para ello, utiliza el valor de una propiedad que es visible por todos los hilos, sus valores posibles son True o False y pueden ser asignados con los métodos *set()* y *clear()* respectivamente.

Por otro lado, también se usa el método *wait()* para detener la ejecución de uno o mas hilos hasta que la propiedad alcance el valor True. Dicho metodo, devuelve el valor de la propiedad y cuenta con un argumento opcional para fijar un tiempo de espera. Otra opción para obtener el estado de un evento es mediante el metodo *is_set()*.

Para ver otro ejemplo de esto, ya que el mostrado antes era considerablemente mas complejo, tenemos:

```
import threading, random

def gen_pares():
    num_pares = 0
    print('Números:', end=' ')
    while num_pares < 25:
        numero = random.randint(1, 10)
        resto = numero % 2
        if resto == 0:
            num_pares +=1
            print(numero, end=' ')
    print()

def contar():
    contar = 0
    nom_hilo = threading.current_thread().getName()
    print(nom_hilo, "en espera")
    estado = evento.wait()
    while contar < 25:
        contar+=1
        print(nom_hilo, ':', contar)

evento = threading.Event()
hilo1 = threading.Thread(name='h1', target=contar)
hilo2 = threading.Thread(name='h2', target=contar)
hilo1.start()
hilo2.start()

print('Obteniendo 25 números pares...')
gen_pares()
print('Ya se han obtenido')
evento.set()
```

Aquí, se inician dos hilos que permanecen a la espera hasta la obtención de 25 números pares (que son los números generados con la función *radint()* del módulo *random*). Luego, cuando se tienen todos los números los dos hilos continúan su ejecución de manera sincronizada.

Otra opción que tenemos es la de sincronizar hilos con objetos **Condition**, usados también para sincronizar la ejecución de varios hilos. En este caso los bloqueos se suelen vincular con unas operaciones que se tienen que realizar antes que otras. Por ejemplo:

```
import threading, random, math

def funcion1(condicion):
    global lista
    print(threading.current_thread().name,
          'esperando a que se generen los números')
    with condicion:
        condicion.wait()
        print('Elementos:', len(lista))
        print('Suma total:', math.fsum(lista))

def funcion2(condicion):
    global lista
    print(threading.current_thread().name,
          'generando números')
    with condicion:
        for numeros in range(1, 1001):
            entero = random.randint(1,100)
            lista.append(entero)
        print('Ya hay 1000 números')
        condicion.notifyAll()

lista = []
condicion = threading.Condition()
hilo1 = threading.Thread(name='hilo1', target=funcion1,
                         args=(condicion,))
hilo2 = threading.Thread(name='hilo2', target=funcion2,
                         args=(condicion,))

hilo1.start()
hilo2.start()
```

Aquí Podemos ver que un hilo espera, llamado al método *wait()*, a que otro hilo genere mil números aleatorios que son añadidos a una lista. Luego, una vez que se han obtenido los números el hecho es notificado con *notifyAll()* al hilo que espera. Finalmente, el hilo detenido continúa su ejecución mostrando el número de elementos generados y la suma de todos ellos, con la función *fsum()* del módulo *math*.

Otra opción con la que contamos es la de sincronizar con objetos **Barrier**, quienes actúan como una barrera que mantiene los hilos bloqueados en un punto del programa hasta que todos hayan alcanzado ese punto. Si vemos un ejemplo básico de su uso:

```
import threading, random, math

def funcion1(barrera):
    nom_hilo = threading.current_thread().name
    print(nom_hilo,
          'Esperando con',
          barrera.n_waiting,
          'hilos más')
```

```
    numero = random.randint(1,10)
    ident = barrera.wait()
    print(nom_hilo,
          'Ejecutando después de la espera',
          ident)
    print('factorial de',
          numero,
          'es',
          math.factorial(numero))

NUM_HILOS = 5
barrera = threading.Barrier(NUM_HILOS)
hilos = [threading.Thread(name='hilo-%s' % i,
                          target=funcion1,
                          args=(barrera,),
                          ) for i in range(NUM_HILOS)]

for hilo in hilos:
    print(hilo.name, 'Comenzando ejecución')
    hilo.start()
```

Tenemos que se inician 5 hilos que obtienen un número aleatorio y permanecen bloqueados en el punto donde se encuentra el método *wait()* a la espera de que el último hilo haya obtenido su número. Después, continúan todos mostrando el factorial del número obtenido en cada caso.

También existe la posibilidad de enviar un aviso de cancelación a todos los hilos que esperan con el método *abort()* del objeto **Barrier**. Esta acción genera una excepción de tipo *threading.BrokenBarrierError* que se deberá capturar y tratar de forma conveniente haciendo algo como por ejemplo:

```
try:
 ident = barrera.wait()
except threading.BrokenBarrierError:
 print(nom_hilo, 'Cancelando')
else:
 print('Ejecutando después de la espera', ident)
```

Y finalmente también podemos mencionar la sincronización usando objetos **Semaphore**, quienes también mencionamos en la pregunta anterior y que pudimos observar que funcionan tal cual como un semáforo. En este caso podríamos imaginarnos un semáforo que esta en la puerta de un estacionamiento y sólo da luz verde para que pasen carros si hay puestos libres y da luz roja e impide el acceso, si se ocupa el estacionamiento, hasta que haya al menos un puesto libre. Este ejemplo nos ayuda a precisar mejor lo que vimos en la pregunta anterior ya que observamos que estos semáforos son usados principalmente para controlar el acceso concurrente de los hilos a una parte del código o aun conjunto de datos.

Como tal estos son las principales primitivas de sincronización que maneja Python.