Creado por:

Isabel Maniega

Threads

Paralelismo basado en hilos.

Thread es una unidad de ejecución que forma parte de un proceso. Un proceso puede tener varios subprocesos y todos ejecutarse al mismo tiempo. Es una unidad de ejecución en programación concurrente. Un hilo es liviano y un programador puede administrarlo de forma independiente. Le ayuda a mejorar el rendimiento de la aplicación mediante el paralelismo.

Diferencia entre proceso y Threads (hilos):

- Proceso significa que un programa está en ejecución, mientras que hilo significa un segmento de un proceso.
- Un proceso no es liviano, mientras que los subprocesos son livianos.
- Un proceso tarda más en finalizar y el hilo tarda menos en finalizar.
- El proceso requiere más tiempo para su creación, mientras que Thread requiere menos tiempo para su creación.
- Es probable que el proceso requiera más tiempo para el cambio de contexto, mientras que Threads requiere menos tiempo para el cambio de contexto.
- Un proceso está mayoritariamente aislado, mientras que los subprocesos comparten memoria.
- El proceso no comparte datos y los subprocesos comparten datos entre sí.

La forma más sencilla de usar un Thread es crear una instancia con un función de destino y llamar a start() para que comience a funcionar.

In [1]:
```
# Importamos la librería threading
import threading


def worker():
    """
        Función con la actividad
        que debe realizar el hilo
    """
    print('Worker')

# listado de hilos creados
threads = []
for i in range(5):
    # Creamos 5 subprocesos distintos y pasamos la función que deben real
    t = threading.Thread(target=worker)
    threads.append(t)
    # Iniciamos el subproceso
```

```
    t.start()
print(threads)
```

```
Worker
Worker
Worker
Worker
Worker
[<Thread(Thread-5 (worker), stopped 132718404830784)>, <Thread(Thread-6 (w
orker), stopped 132718404830784)>, <Thread(Thread-7 (worker), stopped 1327
18404830784)>, <Thread(Thread-8 (worker), stopped 132718404830784)>, <Thre
ad(Thread-9 (worker), stopped 132718404830784)>]
```

La salida son cinco líneas con "Worker" en cada una.

Es útil poder generar un hilo y pasarle argumentos para decirle que trabajo hacer. Cualquier tipo de objeto puede ser pasado como argumento al hilo. Este ejemplo pasa un número, que luego el hilo imprime.

In [2]:
```
# Importamos la librería threading
import threading


def worker(num):
    """
        Función con la actividad
        que debe realizar el hilo
        num: int
    """
    print('Worker: %s' % num)

# listado de hilos creados
threads = []
for i in range(5):
    """
        Creamos 5 subprocesos distintos y
        pasamos la función que deben realizar,
        además los argumentos de la función (args)
    """
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    # Iniciamos el subproceso
    t.start()
print(threads)
```

```
Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4
[<Thread(Thread-10 (worker), stopped 132718404830784)>, <Thread(Thread-11
(worker), stopped 132718404830784)>, <Thread(Thread-12 (worker), stopped 1
32718404830784)>, <Thread(Thread-13 (worker), stopped 132718404830784)>, <
Thread(Thread-14 (worker), stopped 132718404830784)>]
```

El argumento entero ahora está incluido en el mensaje impreso por cada hilo.

Determinar el hilo actual

Usar argumentos para identificar o nombrar el hilo es engorroso e innecesario. Cada instancia Thread tiene un nombre con un valor predeterminado que se puede cambiar cuando se crea el hilo. Nombrar hilos es útil en procesos de servidor con múltiples hilos de servicio manejando diferentes operaciones.

In [5]:
```
import threading
import time


def worker():
    """
        Imprimimos el nombre del actual subproceso
        que esta ejecutándose:
        threading.current_thread().getName()
    """
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(0.2)
    print(threading.current_thread().getName(), 'Exiting')


def my_service():
    """
        Imprimimos el nombre del actual subproceso
        que esta ejecutándose:
        threading.current_thread().getName()
    """
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(0.3)
    print(threading.current_thread().getName(), 'Exiting')

# El argumento name nos permite nombrar el hilo
# Creamos dos subprocesos con funciones distintas
t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
# Subproceso creado con nombre por defecto
w2 = threading.Thread(target=worker)

w.start()
w2.start()
t.start()
```

```
worker Starting
Thread-17 (worker) Starting
my_service Starting
/tmp/ipykernel_29116/1782560158.py:11: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(threading.current_thread().getName(), 'Starting')
/tmp/ipykernel_29116/1782560158.py:22: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(threading.current_thread().getName(), 'Starting')
/tmp/ipykernel_29116/1782560158.py:13: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(threading.current_thread().getName(), 'Exiting')
/tmp/ipykernel_29116/1782560158.py:24: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(threading.current_thread().getName(), 'Exiting')
```

```
worker Exiting
Thread-17 (worker) Exiting
my_service Exiting
```

La salida de depuración incluye el nombre del hilo actual en cada línea. Las líneas con "Thread-X" en la columna de nombre de hilo corresponden al hilo sin nombre w2.

Hilos de Daemon vs. No-Daemon

Hasta este punto, los programas de ejemplo han esperado implícitamente para salir hasta que todos los hilos hayan completado su trabajo. A veces los programas generan un hilo como un demonio que se ejecuta sin bloquear el programa principal de salir. El uso de hilos de demonio es útil para servicios donde puede que no haya una manera fácil de interrumpir el hilo, o donde dejar que el el hilo muera en medio de su trabajo, no pierde ni corrompe los datos (por ejemplo, un hilo que genera «latidos del corazón» para una herramienta de monitoreo de servicio). Para marcar un hilo como demonio, pasa daemon=True al construirlo o llama a su método set_daemon() con True. El valor predeterminado es que los subprocesos no sean demonios.

In [6]:
```
import threading
import time


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


def non_daemon():
    print(threading.current_thread().getName(), 'Starting')
    print(threading.current_thread().getName(), 'Exiting')

"""
    Iniciamos dos subprocesos:
    - name 'daemon' como demonio a True
    - name non_daemon como demonio a False
"""
d = threading.Thread(name='daemon', target=daemon, daemon=True)
t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
```

```
daemon Starting
non-daemon Starting
non-daemon Exiting
```

```
/tmp/ipykernel_29116/390668265.py:6: DeprecationWarning: getName() is deprecated, get the name attribute instead
  print(threading.current_thread().getName(), 'Starting')
/tmp/ipykernel_29116/390668265.py:12: DeprecationWarning: getName() is deprecated, get the name attribute instead
  print(threading.current_thread().getName(), 'Starting')
/tmp/ipykernel_29116/390668265.py:13: DeprecationWarning: getName() is deprecated, get the name attribute instead
  print(threading.current_thread().getName(), 'Exiting')
/tmp/ipykernel_29116/390668265.py:8: DeprecationWarning: getName() is deprecated, get the name attribute instead
  print(threading.current_thread().getName(), 'Exiting')
daemon Exiting
```

La salida no incluye el mensaje "Exiting" del hilo demonio, ya que todos los hilos no demonio (incluyendo el hilo principal) terminan antes de que el hilo demonio se despierte de la llamada sleep().

Para esperar hasta que un subproceso demonio haya completado su trabajo, usa el método join().

In [7]:
```python
import threading
import time


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


def non_daemon():
    print(threading.current_thread().getName(), 'Starting')
    print(threading.current_thread().getName(), 'Exiting')

"""
    Iniciamos dos subprocesos:
    - name 'daemon' como demonio a True
    - name non_daemon como demonio a False
"""
d = threading.Thread(name='daemon', target=daemon, daemon=True)
t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join()
t.join()
```

```
/tmp/ipykernel_29116/738833378.py:6: DeprecationWarning: getName() is deprecated, get the name attribute instead
  print(threading.current_thread().getName(), 'Starting')
/tmp/ipykernel_29116/738833378.py:12: DeprecationWarning: getName() is deprecated, get the name attribute instead
  print(threading.current_thread().getName(), 'Starting')
/tmp/ipykernel_29116/738833378.py:13: DeprecationWarning: getName() is deprecated, get the name attribute instead
  print(threading.current_thread().getName(), 'Exiting')
/tmp/ipykernel_29116/738833378.py:8: DeprecationWarning: getName() is deprecated, get the name attribute instead
  print(threading.current_thread().getName(), 'Exiting')
daemon Starting
non-daemon Starting
non-daemon Exiting
daemon Exiting
```

Por defecto, join() bloquea indefinidamente. También es posible pasar un valor flotante que represente el número de segundos a esperar por el hilo para convertirse en inactivo. Si el hilo no se completa dentro del período de tiempo de espera, join() retorna de todos modos.

In [8]:
```
import threading
import time


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


def non_daemon():
    print(threading.current_thread().getName(), 'Starting')
    print(threading.current_thread().getName(), 'Exiting')

"""
    Iniciamos dos subprocesos:
    - name 'daemon' como demonio a True
    - name non_daemon como demonio a False
"""
d = threading.Thread(name='daemon', target=daemon, daemon=True)
t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join(0.1)
print('d.is_alive()', d.is_alive())
t.join()
```

```
daemon Starting
non-daemon Starting
non-daemon Exiting
d.is_alive() True
```

```
/tmp/ipykernel_29116/2324766611.py:6: DeprecationWarning: getName() is dep
recated, get the name attribute instead
  print(threading.current_thread().getName(), 'Starting')
/tmp/ipykernel_29116/2324766611.py:12: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(threading.current_thread().getName(), 'Starting')
/tmp/ipykernel_29116/2324766611.py:13: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(threading.current_thread().getName(), 'Exiting')
/tmp/ipykernel_29116/2324766611.py:8: DeprecationWarning: getName() is dep
recated, get the name attribute instead
  print(threading.current_thread().getName(), 'Exiting')
daemon Exiting
```

Dado que el tiempo de espera transcurrido es menor que la cantidad de tiempo que el hilo demonio duerme, el hilo sigue «vivo» después de join() retorna.

Señalización Entre Hilos

Aunque el punto de usar múltiples hilos es ejecutar separadamente operaciones al mismo tiempo, hay momentos en que es importante ser capaz de sincronizar las operaciones en dos o más hilos. Los objetos evento son una forma sencilla de comunicarse entre hilos de forma segura. Un Event gestiona una bandera interna que las personas que llaman pueden controlar con los métodos set() y clear(). Otros hilos pueden usar wait() para pausar hasta que se establezca la bandera, bloqueando efectivamente el avance hasta que se permita continuar.

In [14]:
```
import threading
import time


def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    print(f'({threading.current_thread().getName()} ) wait_for_event star
    event_is_set = e.wait()
    print(threading.current_thread().getName(), f'event set: {event_is_se


def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    while not e.is_set():
        print(f'({threading.current_thread().getName()} ) wait_for_event_
        event_is_set = e.wait(t)
        print(threading.current_thread().getName(), f'event set: {event_i
        if event_is_set:
             print(f'({threading.current_thread().getName()} ) processing
        else:
            print(f'({threading.current_thread().getName()} ) doing other

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, arg
t2.start()

print(f'({threading.current_thread().getName()} ) Waiting before calling
```

```
time.sleep(0.3)
e.set()
print(f'({threading.current_thread().getName()} ) Event is set')
```

```
/tmp/ipykernel_29116/4079042566.py:7: DeprecationWarning: getName() is dep
recated, get the name attribute instead
  print(f'({threading.current_thread().getName()} ) wait_for_event startin
g')
/tmp/ipykernel_29116/4079042566.py:15: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(f'({threading.current_thread().getName()} ) wait_for_event_timeout
starting')
/tmp/ipykernel_29116/4079042566.py:30: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(f'({threading.current_thread().getName()} ) Waiting before calling
Event.set()')
(block ) wait_for_event starting
(nonblock ) wait_for_event_timeout starting
(MainThread ) Waiting before calling Event.set()
(MainThread ) Event is setnonblock event set: True
(nonblock ) processing event

block event set: True
/tmp/ipykernel_29116/4079042566.py:33: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(f'({threading.current_thread().getName()} ) Event is set')
/tmp/ipykernel_29116/4079042566.py:17: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(threading.current_thread().getName(), f'event set: {event_is_se
t}')
/tmp/ipykernel_29116/4079042566.py:19: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(f'({threading.current_thread().getName()} ) processing event')
/tmp/ipykernel_29116/4079042566.py:9: DeprecationWarning: getName() is dep
recated, get the name attribute instead
  print(threading.current_thread().getName(), f'event set: {event_is_se
t}')
```

El método wait_for_event_timeout() toma un argumento que representa el número de segundos que el evento espera antes de que se agote el tiempo de espera. Devuelve un booleano indicando si el evento está configurado o no, para que la persona que llama sepa por qué wait() regresó. El método is_set() puede ser usado por separado en el evento sin miedo a bloquear.

En este ejemplo, wait_for_event_timeout() comprueba el estatus del evento sin bloqueo indefinido. El wait_for_event() bloquea en la llamada a wait(), que no regresa hasta que el estado del evento cambie.

Sincronizar hilos

Además de usar Events, otra forma de sincronizar los hilos son a través del uso de un objeto Condition. Porque Condition utiliza un Lock, se puede vincular a un recurso compartido, permitiendo que múltiples hilos esperen a que el recurso sea actualizado. En este ejemplo, los hilos consumer() esperan el Condition que se establezca antes de continuar. El hilo producer() es responsable de establecer la condición y notificar a los otros hilos que pueden continuar.

In [16]:
```
import threading
import time


def consumer(cond):
    """wait for the condition and use the resource"""
    print(f'({threading.current_thread().getName()} ) Starting consumer t
    with cond:
        cond.wait()
        print(f'({threading.current_thread().getName()} ) Resource is ava


def producer(cond):
    """set up the resource to be used by the consumer"""
    print(f'({threading.current_thread().getName()} ) Starting producer t
    with cond:
        print(f'({threading.current_thread().getName()} ) Making resource
        cond.notify_all()


condition = threading.Condition()
c1 = threading.Thread(name='c1', target=consumer, args=(condition,))
c2 = threading.Thread(name='c2', target=consumer, args=(condition,))
p = threading.Thread(name='p', target=producer, args=(condition,))

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

```
/tmp/ipykernel_29116/2153321253.py:7: DeprecationWarning: getName() is dep
recated, get the name attribute instead
  print(f'({threading.current_thread().getName()} ) Starting consumer thre
ad')
(c1 ) Starting consumer thread
(c2 ) Starting consumer thread
(p ) Starting producer thread
(p ) Making resource available
/tmp/ipykernel_29116/2153321253.py:15: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(f'({threading.current_thread().getName()} ) Starting producer thre
ad')
/tmp/ipykernel_29116/2153321253.py:17: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(f'({threading.current_thread().getName()} ) Making resource availa
ble')
/tmp/ipykernel_29116/2153321253.py:10: DeprecationWarning: getName() is de
precated, get the name attribute instead
  print(f'({threading.current_thread().getName()} ) Resource is available
to consumer')
(c1 ) Resource is available to consumer
(c2 ) Resource is available to consumer
```

Los hilos usan with para adquirir el bloqueo asociado con la Condition. Usando los métodos capture() y release() explícitamente también funcionan.

Las barreras son otro mecanismo de sincronización de hilos. Una Barrier establece un punto de control y todos los hilos participantes bloquean hasta que todas las «partes» participantes hayan alcanzado ese punto. Permite que los hilos se inicien por separado y luego se pause hasta que todos están listos para continuar.

In [23]:
```
import threading
import time


def worker(barrier):
    print(threading.current_thread().name, 'waiting for barrier with {} o
    worker_id = barrier.wait()
    print(threading.current_thread().name, 'after barrier', worker_id)


NUM_THREADS = 3

barrier = threading.Barrier(NUM_THREADS)

threads = [threading.Thread(name='worker-%s' % i, target=worker, args=(ba
           for i in range(NUM_THREADS)]

for t in threads:
    print(t.name, 'starting')
    t.start()
    time.sleep(0.1)

for t in threads:
    t.join()
```

```
worker-0 starting
worker-0 waiting for barrier with 0 others
worker-1 starting
worker-1 waiting for barrier with 1 others
worker-2 starting
worker-2 waiting for barrier with 2 others
worker-2 after barrier 2
worker-0 after barrier 0
worker-1 after barrier 1
```

En este ejemplo, la Barrier está configurada para bloquear hasta que tres hilos estén esperando. Cuando se cumple la condición, todos los los hilos se liberan más allá del punto de control al mismo tiempo. Los valores de retorno de wait() indica el número de la parte que está siendo liberada, y puede usarse para limitar algunos subprocesos de realizar una acción como limpiar un recurso compartido.

Colas o Queue (FIFO)

FIFO: primero en entrar, primero en salir

In [24]:
```
import queue

q1 = queue.Queue(5)
q1.put(1)
q1.put(2)
```

```
q1.put(3)
q1.__dict__
```

Out[24]:
```
{'maxsize': 5,
 'queue': deque([1, 2, 3]),
 'mutex': <unlocked _thread.lock object at 0x78b4f48b59c0>,
 'not_empty': <Condition(<unlocked _thread.lock object at 0x78b4f48b59c0
>, 0)>,
 'not_full': <Condition(<unlocked _thread.lock object at 0x78b4f48b59c0
>, 0)>,
 'all_tasks_done': <Condition(<unlocked _thread.lock object at 0x78b4f48
b59c0>, 0)>,
 'unfinished_tasks': 3}
```

In [25]:
```
q1.get()
```

Out[25]:
```
1
```

In [26]:
```
q1.get()
```

Out[26]:
```
2
```

In [27]:
```
q1.__dict__
```

Out[27]:
```
{'maxsize': 5,
 'queue': deque([3]),
 'mutex': <unlocked _thread.lock object at 0x78b4f48b59c0>,
 'not_empty': <Condition(<unlocked _thread.lock object at 0x78b4f48b59c0
>, 0)>,
 'not_full': <Condition(<unlocked _thread.lock object at 0x78b4f48b59c0
>, 0)>,
 'all_tasks_done': <Condition(<unlocked _thread.lock object at 0x78b4f48
b59c0>, 0)>,
 'unfinished_tasks': 3}
```

Compartir datos entre threads

In [28]:
```
import threading
import queue


def busqueda1(num,cola):
    for i in range(5):
        if num == i:
            cola.put("Busqueda1 confirmó el número")

def busqueda2(num,cola):
    for i in range(5,10):
        if(num == i):
            cola.put("Busqueda2 confirmó el número")
numero = 3
cola = queue.Queue()

thread1 = threading.Thread(target=busqueda1,args=(numero,cola))
thread2 = threading.Thread(target=busqueda2,args=(numero,cola))

thread1.start()
thread2.start()
```

```
thread1.join()
thread2.join()

print('Resultado: ', cola.get())
```

```
Resultado:  Busqueda1 confirmó el número
```

In [30]:

```
import queue
import random
import threading
import time

def producer(queue, event):
    """Pretend we're getting a number from the network."""
    while not event.is_set():
        message = random.randint(1, 10)
        print("%s got message: %s" % (threading.current_thread().name, me
        queue.put(message)
        event.wait()

    print(f"{threading.current_thread().name} received event. Exiting")

def consumer(queue, event):
    """Pretend we're saving a number in the database."""
    print('Valores en la cola: ', list(queue.queue))
    while not event.is_set() or not queue.empty():
        message = queue.get()
        print("%s storing message: %s (size=%d)" % (threading.current_thr
                                                    message, queue.qsize(
        event.wait()

    print(f"{threading.current_thread().name} received event. Exiting")

cola = queue.Queue()
event = threading.Event()

thread1 = threading.Thread(name='Producer1', target=producer,args=(cola,
thread3 = threading.Thread(name='Producer2', target=producer,args=(cola,
thread2 = threading.Thread(name='Consumer1',target=consumer,args=(cola, e
thread4 = threading.Thread(name='Consumer2',target=consumer,args=(cola, e

thread1.start()
thread3.start()
thread2.start()
thread4.start()

time.sleep(0.1)
print("Main: about to set event")
event.set()
```

```
Producer1 got message: 4
Producer2 got message: 5
Valores en la cola:  [4, 5]
Consumer1 storing message: 4 (size=1)
Valores en la cola:  [5]
Consumer2 storing message: 5 (size=0)
Main: about to set event
Consumer1 received event. Exiting
Producer1 received event. Exiting
Producer2 received event. Exiting
Consumer2 received event. Exiting
```

Creado por:

Isabel Maniega