Threads

December 12, 2020

1 Hilos

Es una pieza de código, muy pequeña y compacta que puede ejecutar un sistema operativo. Los hilos de ejecución comparten recursos, como memoria, archivos abiertos, etc.

Un hilo es un tarea que puede ser "ejecutada al mismo tiempo" que otra tarea. Los hilos pueden tener diferentes estados: "Ejecutando", "Listo" y "bloqueado". Los hilos suelen estar dentro de los procesos.

El multi-threading es una técnica que permite desacoplar las tareas que no son secuencialmente dependientes.

Hay dos tipos de hilos: * en el espacio de kernel * en el espacio de usuario

1.1 Ventajas del multithread

- puede correr más rapido en computadoras con múltiples CPUs
- Los hilos de un proceso comparten memoria. Y pueden compartir variables globales.

Para manejar hilos en Python tenemos el módulo *threading*

2 Threading

En python existe el objeto Thread, que es una clase que representa un hilo de control. Se puede crear de dos maneras, pasando una función (callable object) o sobreescribiendo `run()`. Cuando se hereda de Thread solo se debe sobreeescribir `__init__(self)` y `run()`.

Para que la ejecución del hilo comience se debe llamar al método `start()`, este invoca a `run()`. Se puede utilizar el método `is_alive()` para saber si un hilo está vivo o no.

El método `join()` bloquea la llamada al hilo que termina.

[3]:
```
import time
from threading import Thread

def sleeper(i):
    print (f"thread {i} sleeps for 5 seconds")
    time.sleep(5)
    print (f"thread {i} woke up")

for i in range(10):
```

```
    t = Thread(target=sleeper, args=(i,))
    t.start()

t.join()
```

[4]:
```
import threading
import time
import logging
import random

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-9s) %(message)s',)

class Counter(object):
```

```
    def __init__(self, start = 0):
        self.lock = threading.Lock()
        self.value = start
    def increment(self):
        logging.debug('Waiting for a lock')
        self.lock.acquire()
        try:
            logging.debug('Acquired a lock')
            self.value = self.value + 1
        finally:
            logging.debug('Released a lock')
            self.lock.release()

def worker(c):
    for i in range(2):
        r = random.random()
        logging.debug('Sleeping %0.02f', r)
        time.sleep(r)
        c.increment()
    logging.debug('Done')

if __name__ == '__main__':
    counter = Counter()
    for i in range(2):
        t = threading.Thread(target=worker, args=(counter,))
        t.start()

    logging.debug('Waiting for worker threads')
    main_thread = threading.currentThread()
    for t in threading.enumerate():
        if t is not main_thread:
            t.join()
    logging.debug('Counter: %d', counter.value)
```

```
(Thread-18) Released a lock
(Thread-18) Sleeping 0.51
(Thread-18) Waiting for a lock
(Thread-18) Acquired a lock
(Thread-18) Released a lock
(Thread-18) Done
```

```
      ␣
↪------------------------------------------------------------------------
```

```
       KeyboardInterrupt                         Traceback (most recent call␣
↪last)

       <ipython-input-4-8695e64159da> in <module>
        39     for t in threading.enumerate():
        40         if t is not main_thread:
   ---> 41             t.join()
        42     logging.debug('Counter: %d', counter.value)


       /usr/lib/python3.8/threading.py in join(self, timeout)
      1009
      1010         if timeout is None:
   -> 1011             self._wait_for_tstate_lock()
      1012         else:
      1013             # the behavior of a negative timeout isn't documented,␣
↪but


       /usr/lib/python3.8/threading.py in _wait_for_tstate_lock(self, block,␣
↪timeout)
      1025         if lock is None:  # already determined that the C code is␣
↪done
      1026             assert self._is_stopped
   -> 1027         elif lock.acquire(block, timeout):
      1028             lock.release()
      1029             self._stop()


       KeyboardInterrupt:
```

2.2 Queue

Las colas es otra manera de lograr sincronización entre procesos. Es muy util cuando los hilos necesitan intercambiar información de manera segura.

Más detalles: https://docs.python.org/es/3/library/queue.html#module-queue

[5]:
```python
import threading, queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

def worker2():
    # hay logica
    worker(dinero)


# turn-on the worker thread
threading.Thread(target=worker, daemon=True).start()

# send thirty task requests to the worker
for item in range(30):
    q.put({"id": 2, "otra": "hola"})

print('All task requests sent\n', end='')

# block until all tasks are done
q.join()
print('All work completed')
```

```
All task requests sent
Working on 0
Finished 0
Working on 1
Finished 1
Working on 2
Finished 2
Working on 3
Finished 3
Working on 4
Finished 4
Working on 5
Finished 5
Working on 6
Finished 6
Working on 7
Finished 7
Working on 8
```

```
Finished 8
Working on 9
Finished 9
Working on 10
Finished 10
Working on 11
Finished 11
Working on 12
Finished 12
Working on 13
Finished 13
Working on 14
Finished 14
Working on 15
Finished 15
Working on 16
Finished 16
Working on 17
Finished 17
Working on 18
Finished 18
Working on 19
Finished 19
Working on 20
Finished 20
Working on 21
Finished 21
Working on 22
Finished 22
Working on 23
Finished 23
Working on 24
Finished 24
Working on 25
Finished 25
Working on 26
Finished 26
Working on 27
Finished 27
Working on 28
Finished 28
Working on 29
Finished 29
All work completed
```

2.3 Python GIL (Python Global Interpreter Lock)

Hace que en cualquier instante tiempo siempre existe uno y nada más que un hilo ejecutandose. Por lo que es imposible, hacer uso de multiple procesos con hilos. No todo es malo.

Que hace GIL:

- Limita las operaciones de hilos
- Ejecuciones paralelas están restringidas.
- Se asegura que un hilo se ejecute por vez.
- Simplifica tener que preocuparnos por detalles de memoria.

TODO: 1. multiprocessing 2. async