Python Threads

1. Introduzione

Questa guida è rivolta a chi conosce già le basi teoriche della programmazione multithreading, e vuole imparare i meccanismi messi a disposizione dal linguaggio Python.

2. Creazione di un primo thread

Cominciamo con un esempio pratico:

```
import threading
import time
import random

class ExampleThread(threading.Thread):

  def __init__(self, n):
    threading.Thread.__init__(self)
    self.id = n

  def run(self):
    print "Thread " + str(self.id) + " running..."
    secs = random.randint(3, 7)
    time.sleep(secs)
    print "Thread " + str(self.id) + " exiting..."

threads = []
N_THREADS = 3

for i in range(1, N_THREADS+1):
  t = ExampleThread(i)
  threads.append(t)

for i in range(1, N_THREADS+1):
  threads[i-1].start()
```

`threading` è il modulo privilegiato per creare e gestire i thread. Ogni thread è rappresentato da una classe che estende la classe `Thread` e ridefinisce il suo metodo `run()`. Questo metodo è lo starting point del thread. Le due chiamate a `randint`() e `sleep`() diminuiscono un po' la prevedibilità dell'esecuzione.
Nel "main", si creano diversi oggetti di tipo ExampleThread: l'esecuzione vera e propria del thread parte quando viene chiamato il metodo `start()`.
La chiamata al costruttore di `Thread` è obbligatoria: tramite il costruttore è possibile ridefinire alcune proprietà del thread come il nome o il gruppo di appartenenza.

Il thread è posto in stato attivo alla chiamata a `start()` e vi resta finché non termina il metodo `run()` o viene lanciata un'eccezione non gestita. Il programma termina solo quando tutti i thread sono terminati. È possibile far sì che un thread non sia "conteggiato" (e quindi continui la propria esecuzione anche se tutti gli altri thread sono terminati facendo terminare il programma) impostando `self.daemon = True` nel thread.

Due altri metodi utili:
`threading.enumerate()`: restituisce una lista di oggetti thread attivi (compresi i demoni).
`thread.currentThread()`: restituisce l'oggetto thread correntemente in esecuzione, `thread` è

un secondo modulo più a basso livello.

3. Join

Come in altre librerie, è possibile per un thread t1 “joinare” un altro thread t2 chiamando

```
t2.join()
```

all'interno di t1. Il thread t1 sarà posto in stato di sleep finché non sarà risvegliato dalla terminazione di t2 e continuerà la propria esecuzione. È importante notare che durante lo stato di sleep un thread non è schedulato e quindi non spreca risorse di CPU “non facendo niente”. Una seconda versione del metodo `join()` permette di passare come argomento il numero massimo di secondi da aspettare, dopo i quali l'esecuzione continua lo stesso.

4. Lock

Il python dispone di due metodi per usare i lock: un lock primitivo e un lock più sofisticato detto lock rientrante o rlock. La differenza più importante è che il rlock è effettivamente posseduto da un oggetto e quindi può essere acquisito più volte in modo annidato dallo stesso thread.
Il funzionamento è molto semplice. Mettiamo di avere diversi thread che eseguono lo stesso metodo `run`, e questo metodo contiene una porzione di codice “pericolosa”, ovvero che compie un'operazione che ha bisogno di sincronizzazione con gli altri thread e non è atomica. Ciò che si farà sarà dichiarare nell'oggetto un thread un oggetto di tipo Lock tramite

```
vlock = threading.Lock()
```

e la parte “pericolosa” di codice sarà racchiusa fra le due chiamate

```
vlock.acquire()
# codice non thread-safe
vlock.release()
```

La chiamata ad `acquire()` è bloccante: il thread si blocca finché non riceve il permesso di proseguire (e se ci sono diversi thread in blocco su questa istruzione, non sappiamo chi prenderà per primo l'esecuzione appena il lock è di nuovo disponibile); dopo aver eseguito il codice, rilascia il lock in modo che altri thread in attesa possano usarlo.
È possibile con gli rlock specificare un comportamento non bloccante passando come primo parametro di acquire il valore False: in questo caso la chiamata restituirà subito True se il lock è stato acquisito (ad esempio perché non c'era nessun altro ad usarlo o contenderlo), False altrimenti.

5. Condition variables

Una condizione molto tipica nella programmazione concorrente è che un thread prima di proseguire con una porzione di codice abbia bisogno di attendere che un altro thread abbia compiuto qualche operazione: ad esempio potrebbe aspettare che il thread termini di scrivere dati su un file prima di procedere alla lettura.
La classe che realizza questo in python è `threading.Event`, in realtà un wrapper di alto livello per `threading.Condition`. Si crea un oggetto di tipo Event con

```
event = threading.Event()
```

A questo punto la chiamata a

```
event.wait()
```

metterà il thread corrente in stato di sleep, mentre

```
event.set()
```

invierà a tutti i thread in attesa tramite `wait()` il segnale di risveglio. È importante che prima di chiamare una wait si chiami il metodo `clear()`: questo metodo “cancella” eventuali precedenti occorrenze dell'evento, altrimenti la chiamata di `wait()` potrebbe leggere il segnale inviato precedentemente come inviato a se stessa e ritornerebbe subito.

Lisa “shainer” Vitolo