Programmazione Concorrente in Python

Advanced Computer Programming

Prof. Luigi De Simone

Sommario

- **Argomenti**
  - Multithreading in Python
    - Threading module
  - Global Interpreter Lock
  - Multiprocessing in Python
    - I moduli *multiprocessing* e *multiprocess*

- **Riferimenti**
  - https://docs.python.org/3/library/threading.html#
  - https://realpython.com/python-gil/
  - https://docs.python.org/3/library/multiprocessing.html

Concurrency Programming in Python

- Principalmente, Python mette a disposizione **due moduli** per la **programmazione concorrente**:
  - **`threading`: permette la creazione di software **multi-thread****
  - **`multiprocessing`: permette la creazione di software **multi-process****

- Entrambi i moduli mettono a disposizione i meccanismi necessari a supportare l'esecuzione di **codice concorrente**

Il modulo `threading`

Il modulo **`threading`** mette a disposizione un'**interfaccia di alto livello** per il multi-threading, costruita a partire dal modulo **`_thread`**

- `_thread` mette a disposizione **primitive di basso livello** per l'avvio e la sincronizzazione di thread, ma il suo utilizzo è deprecato
- Mette a disposizione un insieme di classi
  - `Thread`
  - `Thread-local Data`
  - `Lock/RLock`
  - `Condition`
  - `Semaphore`
  - `Event`

`Thread` class

- La classe **Thread** rappresenta un'attività che esegue in un flusso di controllo separato (un thread)
- Esiste un oggetto "**main thread**" che corrisponde al flusso di controllo associato all'intero programma Python
- Esistono vari modi per creare thread in Python, due dei quali sono:

Istanziare un oggetto Thread, passando un "callable object" al costruttore della classe

```
import threading

def func():
    print("Thread running")


if __name__ == "__main__":

    # creating thread
    t1 = threading.Thread(target=func)

    # starting thread
    t1.start()

    # wait until the thread finishes
    t1.join()
```

Creare una classe che estende quella Thread, ridefinendo il metodo run()

```
import threading

class myThread(threading.Thread):

    def run(self):
        print("Thread running")


if __name__ == "__main__":

    # creating thread
    t1 = myThread()

    # starting thread
    t1.start()

    # wait until the thread finishes
    t1.join()
```

`Thread` class

Alcuni dei metodi messi a disposizione dalla **classe Thread** sono:

- **`Thread`**`(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None):`
  - Costruttore della classe Thread
    - *group*, riservato per future implementazioni di ThreadGroup. Deve essere **None**
    - *target*, rappresenta il “callable object” che sarà invocato dal metodo *run*
    - *name*, nome del thread
    - *args*, tupla contenente gli argomenti da passare all’invocazione del *target*
    - *kwargs*, dizionario di keyword da passare all’invocazione del *target*
    - *daemon*, boolean che specifica se il thread deve essere un demone o meno

- **`start()`**
  - Avvia il thread, portando all’esecuzione del metodo *run* non appena il thread sarà schedulato

- **`run()`**
  - Metodo che contiene il corpo del thread, e che ne rappresenta il comportamento

`Thread` class

- **`join(timeout=None)`**
  - Metodo bloccante, che attende fino alla terminazione (naturale o con eccezioni) del thread.
    - *timeout*, numero floating point utilizzato per definire il tempo massimo di attesa (in secondi)

- **`is_alive()`**
  - Metodo che ritorna se il Thread è alive o meno.
  - Dovrebbe essere utilizzato in combinazione con il metodo *join* quando viene specificato un timeout; in questo caso infatti *join* ritorna sempre None, anche nel caso di *timeout* scaduto.

daemon thread vs *non-daemon* thread

- Un thread può essere contrassegnato come "thread demone"
- Il significato di questo flag è che l'intero programma Python termina quando rimangono solo thread di tipo daemon
  - In altre parole, il programma non può terminare se c'è almeno un *thread non-daemon ancora vivo*
- Il valore iniziale, per la proprietà *daemon*, è ereditato dal thread da cui si sta creando il nuovo
- Come abbiamo visto, il flag *daemon* può essere impostato tramite l'argomento del costruttore, oppure tramite la proprietà *daemon*

daemon thread vs *non-daemon* thread

- I thread *daemon* vengono **interrotti bruscamente** alla terminazione del processo
- Le loro risorse (come file aperti, transazioni di database, ecc.) potrebbero **non essere rilasciate correttamente**
- Se si vuole che i propri thread si arrestino gradatamente, bisogna renderli *non-daemon* e utilizzare un meccanismo di segnalazione adeguato, come un evento

daemon thread vs *non-daemon* thread: Esempio

Non-daemon threads

```
# import module
from threading import *
import time

# creating a function
def thread_1():
  for i in range(5):
    print('this is non-daemon thread')
    time.sleep(2)

# creating a thread T
T = Thread(target=thread_1)

# starting of thread T
T.start()

# main thread stop execution till 5 sec.
time.sleep(5)
print('main Thread execution’)
```

Output:
this is non-daemon thread
this is non-daemon thread
this is non-daemon thread
main Thread execution
this is non-daemon thread
this is non-daemon thread

Daemon threads

```
# import modules
from threading import *
import time

# creating a function
def thread_1():
  for i in range(5):
    print('this is thread T')
    time.sleep(3)

# creating a thread
T = Thread(target = thread_1)

# change T to daemon
T.setDaemon(True)

# starting of Thread T
T.start()
time.sleep(5)
print('this is Main Thread')
```

Output:
this is thread T
this is thread T
this is Main Thread

Thread-local Data: the local class

```
import threading
import logging
import random

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-0s)
                    %(message)s',)

def show(d):
    try:
        val = d.val
    except AttributeError:
        logging.debug('No value yet')
    else:
        logging.debug('value=%s', val)

def f(d):
    show(d)
    d.val = random.randint(1, 100)
    show(d)

if __name__ == '__main__':
    d = threading.local()
    show(d)
    d.val = 999
    show(d)

    for i in range(2):
        t = threading.Thread(target=f, args=(d,))
        t.start()
```

- Può essere necessario memorizzare dati unici per ogni thread
- Python fornisce un oggetto thread-local che può essere usato per memorizzare dati unici per ogni thread
- La classe **local** consente di poter gestire dati locali ad un thread:

- creare un'**istanza della classe local** attraverso threading
- **salvare gli attributi** che si vuole rendere locali al thread su tale istanza

Tutti i dati storicizzati nell'istanza local saranno differenti per ogni thread

```
(MainThread) No value yet
(MainThread) value=999
(Thread-1) No value yet
(Thread-1) value=51
(Thread-2) No value yet
(Thread-2) value=19
```

Lock class

La classe **Lock** implementa la rispettiva primitiva di sincronizzazione di basso livello fornita dal modulo *_thread*.
Un lock può trovarsi in due stati: **locked** o **unlocked**

- I lock sono **creati** nello stato ***unlocked***
- Quando un **thread acquisisce un lock**, il lock passa in stato ***locked***
  - successive acquisizioni dello stesso lock da altri thread, porteranno i thread ad essere posti in attesa
- Quando un **thread rilascia un lock** precedentemente acquisito, il lock passa in stato ***unlocked***
  - uno degli eventuali thread in attesa sul lock sarà risvegliato ed eseguito: la scelta del thread varia a seconda delle implementazioni Python

La classe Lock prevede due metodi principali per **l'acquisizione** e **rilascio** di **lock**:

- **acquire()**:
  - Quando lo stato del lock è unlocked(), il metodo cambio lo stato in locked e ritorna immediatamente
  - Quando lo stato è locked. il metodo blocca il thread chiamante fino a quando un altro thread non invoca release() per cambiare lo stato in unlocked
- **release():**
  - Può essere invocata solo quando lo stato del Lock è locked, e porta al passaggio del Lock nello stato unlocked

`Lock` class

- **acquire(blocking=True, timeout=- 1):** acquisisce un lock
  - ritorna `True` se il lock è acquisito, `False` in caso contrario
  - **blocking:** definisce se la chiamata è bloccante
    - True (default), la chiamata è bloccante se il lock è già stato precedentemente acquisito
    - False, la chiamata ritorna immediatamente restituendo False
  - **timeout:** imposta un tempo massimo di attesa
    - se *timeout* (valore floating point) è positivo (default -1), il chiamante resterà bloccato per al massimo *timeout* secondi
    - Allo scadere del timeout, il metodo restituirà `False`
- **release():** rilascia un lock e ritorna (non prevede un valore di ritorno)
- **locked():** verifica lo stato di un Lock
  - se lo stato è locked, ritorna True

Lock class: un esempio

```
from threading import Thread, Lock

counter = 0

def increase(by, lock):
    global counter
    lock.acquire()

    local_counter = counter
    local_counter += by
    counter = local_counter
    print(f'counter={counter}')

    lock.release()

if __name__ == '__main__':

    lock = Lock()

    # create threads
    t1 = Thread(target=increase, args=(10, lock))
    t2 = Thread(target=increase, args=(20, lock))

    # start the threads
    t1.start()
    t2.start()

    # wait for the threads to complete
    t1.join()
    t2.join()

    print(f'The final counter is {counter}')
```

```
counter=10
counter=30
The final counter is 30
```

`RLock` class

RLock implementa i **reentrant lock***:* lock che possono essere acquisiti più volte dallo stesso thread

Oltre al concetto di stato *locked/unlocked*, i reentrant lock prevedono **due ulteriori concetti**:

- **owning thread: quale thread che ha acquisito il lock**
- **recursion level:** per tener traccia del **numero di acquisizioni fatte dallo stesso thread**

Come la classe Lock, sono previsti i metodi **acquire()** e **release()** per modificare lo stato.
Due principali differenze:

- se la `acquire()` è invocata da un thread che già ha acquisito il lock (*owning thread*), **la chiamata non è bloccante** e viene incrementato il *recursion level*
- la `release()` rilascia il lock e **decrementa il *recursion level***
  - solo quando il *recursion level* diventa pari a zero, il lock passa allo stato *unlocked*, e potrà essere effettivamente acquisito da un altro thread

`Condition` class

La classe **Condition** implementa una **condition variable**: permettono ad uno o più thread di rimanere in attesa (che una determinata condizione si verifichi) fino a quando non saranno notificati da un altro thread (quando la condizione è verificata)

Una condition variable è sempre caratterizzata da un *lock*:

- creato automaticamente con la condition variable (default): **il lock creato sarà un *RLock***
- possibilità di utilizzare un lock esistente (*Lock* o *RLock*), passandolo al costruttore della classe
- la classe Condition fornisce i metodi **acquire()** e **release()** che agiscono sul lock a cui la condition variabile fa riferimento

`Condition` class

- **Condition(lock=None):** costruttore della classe
  - **lock:** rappresenta il lock da associare alla condition variable. Se impostato a `None` (default), un RLock sarà creato automaticamente
- **acquire():** chiama il metodo *acquire()* del lock associato alla condition variable
- **release():** chiama il metodo *release()* del lock associato alla condition variable
- **wait(timeout=None):** Attende fino alla notifica, o fino alla scadenza di un timeout.
  - timeout: imposta un tempo massimo di attesa in secondi (floating point). Default a `None`.
  - il metodo rilascia il lock, e resta bloccato fino alla chiamata della *notify()* o *notify_all()* da parte di un altro thread sulla stessa CV, o fino alla scadenza del timeout
  - il lock viene ri-acquisito non appena il thread è risvegliato, ed il metodo ritorna restituendo un booleano
    - False, nel caso di timeout scaduto
    - True, in tutti gli altri casi
- **wait_for(predicate, timeout=None):** attende fino a quando una condizione diventa True
  - **predicate:** deve essere una condizione o un callable object che ritorni un booleano
  - **timeout:** massimo tempo di attesa in secondi
- **notify(n=1):** Risveglia il numero di thread indicati dal parametro n (default 1) in attesa sulla CV
  - **NOTA:** il thread risvegliato non potrà riprendere la sua esecuzione se non riuscirà ad acquisire il lock
- **notify_all():** Risveglia tutti i thread in attesa sulla CV

`Condition` class: un esempio

```
. . .
cv_lock = threading.Lock()
cv = Condition(lock=cv_lock)

. . .

# Consume one item

with cv:

    while not an_item_is_available(queue):

        cv.wait()

    get_an_available_item(queue)



# Produce one item

with cv:

    make_an_item_available(queue)

    cv.notify()
```

NOTA: Il ciclo while che verifica la condizione è necessario

- l'utilizzo di **`with cv`**, evita la necessità di dover richiamare
  - **cv.acquire()** all'inizio della sezione critica
  - **cv.release()** la sezione critica
- quando il thread viene risvegliato dalla notify, la condizione su cui era in attesa potrebbe non essere più verificata
  - Un'alternativa sarebbe utilizzare il metodo **wait_for()**

```
# Consume one item

with cv:

    cv.wait_for(an_item_is_available)

    get_an_available_item()
```

`Semaphore` class

La classe **`Semaphore`** permette di istanziare un oggetto semaforo, il quale gestisce un contatore interno:

- inizializzato alla creazione del semaforo
- decrementato ad ogni operazione di **`acquire()`**
- incrementato ad ogni operazione di **`release()`**

Quando il contatore arriva al valore zero, come conseguenza di operazione di *acquire()*, il thread chiamante viene bloccato fino a quando non viene effettuata una operazione di *release()* da un altro thread sullo stesso oggetto semaphore.

`Semaphore` class

Le primitive fornite dalla classe **Semaphore** sono le seguenti:

- **`class threading.Semaphore(value=1)`:** costruttore della classe
  - **value:** rappresenta il valore a cui è inizializzato il semaforo (default 1)

- **`acquire(blocking=True, timeout=None)`:** acquisisce un semaforo
  - se invocato senza parametri:
    - *se il valore del contatore è maggiore di zero*, decrementa il valore di 1 e ritorna `True`
    - *se il valore del contatore è zero*, la chiamata è bloccante per il thread che l'ha invocata il thread sarà risvegliato in seguito ad una `release()`; al risveglio, il contatore (se maggiore di zero) sarà decrementato, e sarà ritornato `True`
  - **blocking:** se settato a `False`, la chiamata non è bloccante. Pertanto, ritorna `False` immediatamente invece di bloccare il chiamante.
  - **timeout:** rappresenta il massimo tempo (in secondi) per cui il chiamante resta bloccato sulla acquire. Se il timeout scade, il metodo ritorna `False`

- **`release(n=1)`:** rilascia un semaforo, incrementando di un valore pari a *n* il contatore (default n=1). Un numero di thread pari a *n* viene ad essere risvegliato.

Semaphore class: un esempio

```
from threading import *
from time import sleep
from random import random

# creating thread instance where count = 3
obj = Semaphore(3)

def display(name):

    obj.acquire()

    value = random()
    sleep(value)
    print(f'Thread {name} got {value}')

    obj.release()

if __name__ == '__main__':

    threads = []

    # creating and starting multiple threads
    for i in range(10):
        t = Thread(target = display , args = ('Thread-' + str(i),))
        thread.append(t)
        t.start()

    # wait for the threads to complete
    for thread in threads:
        thread.join()
```

```
Thread 1 got 0.4468840323081351
Thread 0 got 0.7288038062917327
Thread 2 got 0.4497887327563145
Thread 4 got 0.019601471581036645
Thread 3 got 0.5114751539092154
Thread 6 got 0.6191428550062478
Thread 5 got 0.9893921843198458
Thread 8 got 0.022640379341017924
Thread 7 got 0.20649643092073067
Thread 9 got 0.18636311846540998
```

`Event` class

- La classe **`Event`** fornisce un semplice **meccanismo di comunicazione tra thread**:
  - Attraverso un **`Event`**, un thread può segnalare un evento per il quale altri thread sono in attesa
  - Ogni **`Event`** è caratterizzato da un flag interno (booleano) che può essere settato a `True` con il metodo `set()`, e resettato a `False` con il metodo `clear()`
- Un thread che è in attesa di un determinato evento, può utilizzare il metodo wait() sull'evento
  - Il thread resterà bloccato fino a quando il flag dell'evento sarà settato a True.

`Event` class

I metodi della classe **Event** sono i seguenti:

- **`is_set()`:** Ritorna `True` se il flag interno è settato a tale valore; `False` in caso contrario.
- **`set()`:** setta il flag interno al valore `Ture`; tutti i thread in attesa dell'evento, vengono risvegliati
- **`clear()`:** setta il flag interno al valore `False`; tutti i thread che invocheranno una wait sull'evento saranno bloccati
- **`wait(timeout=None)`:** pone il thread in attesa che il flag interno venga settato a `True`
  - se il flag è già `True`, ritorna immediatamente
  - in caso contrario, pone il thread in attesa fino al timeout (se settato)
  - **timeout:** valore floating point che rappresenta l'attesa massima in secondi sul metodo wait (default None) Alla scadenza del timeout, il metodo ritorna `False`

`Event` class: un esempio

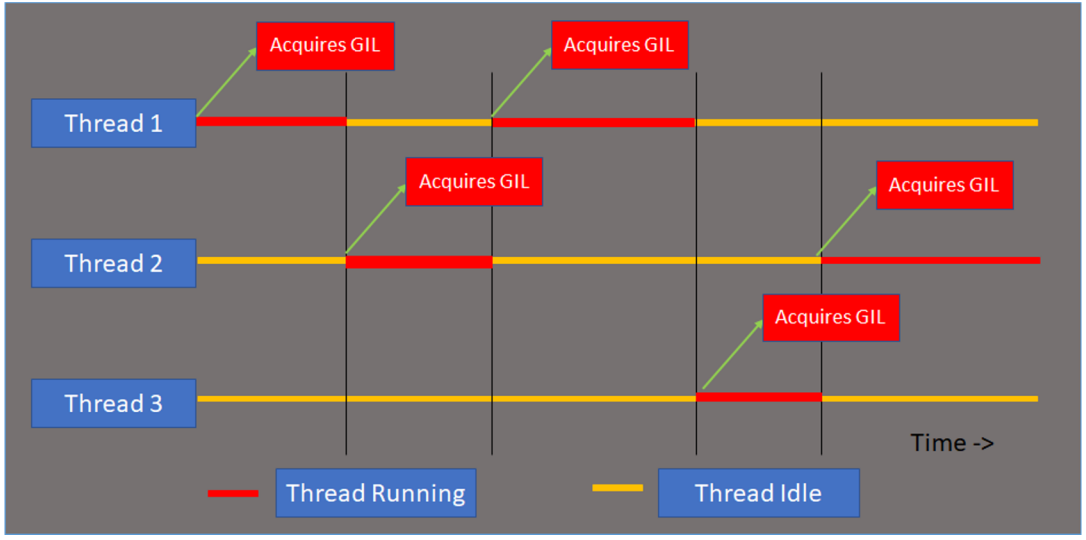
```
import threading as thd
import time

def task(event, timeout):
   print("Started thread but waiting for event...")

   # make the thread wait for event with timeout set
   event_set = event.wait(timeout)

   if event_set:
     print("Event received, releasing thread...")
   else:
     print("Time out, moving ahead without event...")

if __name__ == '__main__':

    # initializing the event object
    e = thd.Event()

    # starting the thread
    thread = thd.Thread(name='Event-Thread', target=task, args=(e,4))
    thread.start()

    # sleeping the main thread for 3 seconds
    time.sleep(3)

    # generating the event
    e.set()
    print("Event is set.")

    # wait for the thread to complete
    thread.join()
```

```
Started thread but waiting for event...
Event is set.
Event received, releasing thread...
```

Multi-threading: Global Interpreter Lock (GIL)

Meccanismo utilizzato dall'interprete Python per assicurare che **un solo thread alla volta possa eseguire bytecode**

- Semplifica l'implementazione dell'interprete
- Rende l'object model Python implicitamente "safe" da accessi concorrenti
- Nato per realizzare thread-safe memory management, richiesto dalle librerie C utilizzate da Python



Multi-threading: Global Interpreter Lock

- **Vantaggi**
  - Permette di rendere facilmente multi-threaded l'interprete Python, a scapito del parallelismo
  - Dovendo gestire un solo lock, migliora le performance degli applicativi single-thread
  - Le librerie C non thread-safe possono essere facilmente integrate
- **Svantaggi**
  - Riduce il livello di parallelismo ottenibile su macchine multi-processore
  - Impatta principalmente i task CPU-bound, in quanto è rilasciato per task I/O-bound

Multi-threading: Still GIL?

- Alcuni moduli (sia standard che di terze parti) sono progettati per **rilasciare il GIL quando vengono eseguiti task che includono *operazioni di I/O bloccanti* (e.g., `time.sleep()`)**

- **Tentativi passati di rimuovere il GIL non hanno avuto successo**
  - Riduzione delle performance degli applicativi single-thread
  - Aumento della complessità dell'interprete
  - Necessità di modificare tutte le estensioni delle librerie C che sfruttano il GIL

Multiprocessing in Python

- Il modulo **`multiprocessing`** supporta lo spawning di processi attraverso una API simile a quella messa a disposizione dal modulo *threading*

- Supera il limite del *threading* dovuto al *GIL*, attraverso l'utilizzo di processi multipli:
  - permette la creazione di **più *Python interpreter processes***
  - **ogni processo avrà un proprio GIL**, evitando i problemi del modulo *threading*
  - permette al programmatore di **sfruttare totalmente le architetture multicore**

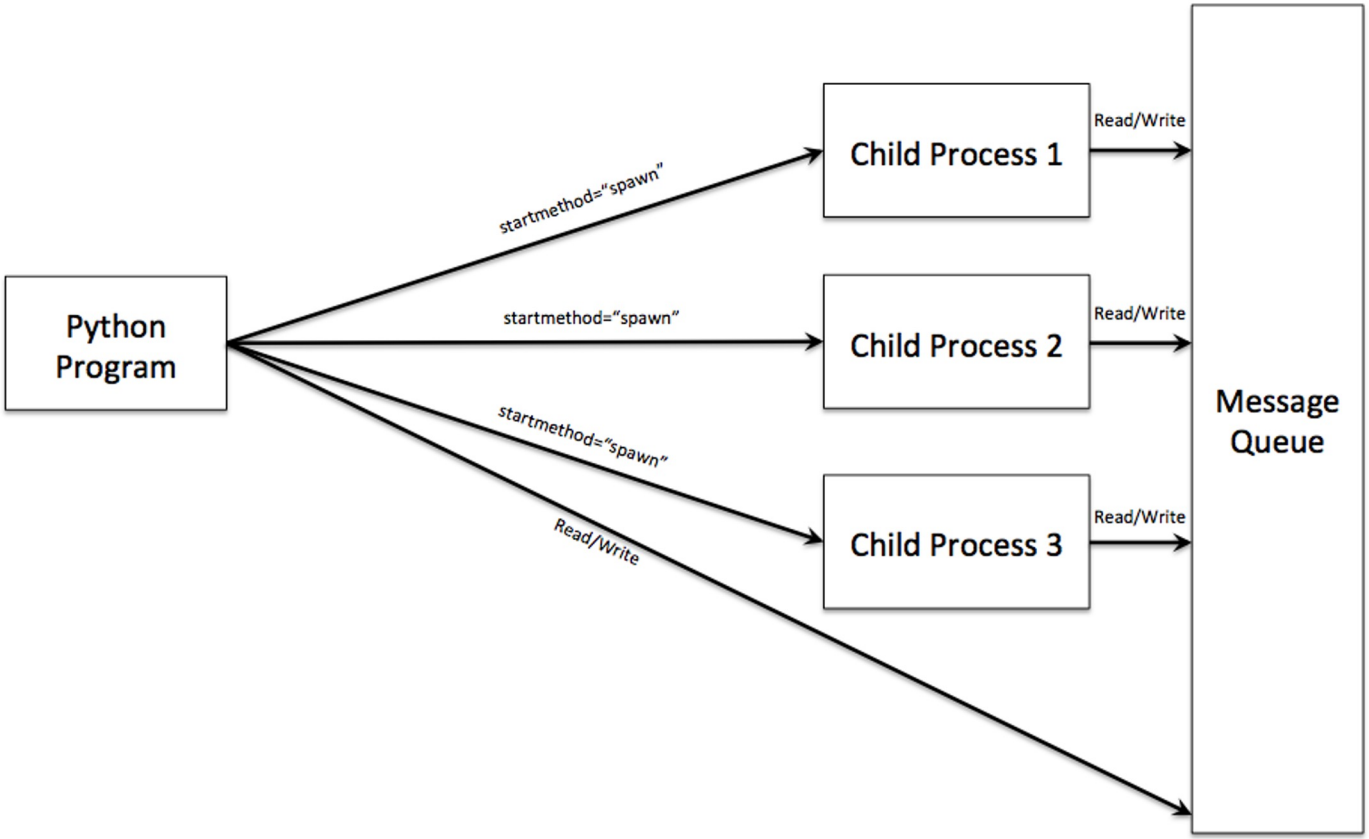
- **E' consigliabile utilizzare questo modulo quando:**
  - si hanno a disposizione core multipli
  - i task da eseguire sono principalmente CPU bound

Multiprocessing in Python

- A seconda della piattaforma, il modulo **`multiprocessing`** supporta **tre modi per avviare un processo attraverso il set di *start_method***
- Per settare lo *start_method* è possibile utilizzare il metodo **`set_start_method(string)`** messo a disposizione nel modulo multiprocessing, passando una stringa con il modo di avvio)
- I metodi di avvio sono **spawn**, **fork**, e forkserver

1. **`spawn`** (disponibile su Unix e Windows, ed il default per Windows e macOS)
    - Il processo padre **avvia un nuovo *Python interpreter process.***
    - Il processo figlio **eredita solo le risorse necessarie ed esegue il metodo run()**, escludendo informazioni non necessarie (ad esempio file descriptor non necessari).
    - **Metodo più lento** dei tre disponibili.

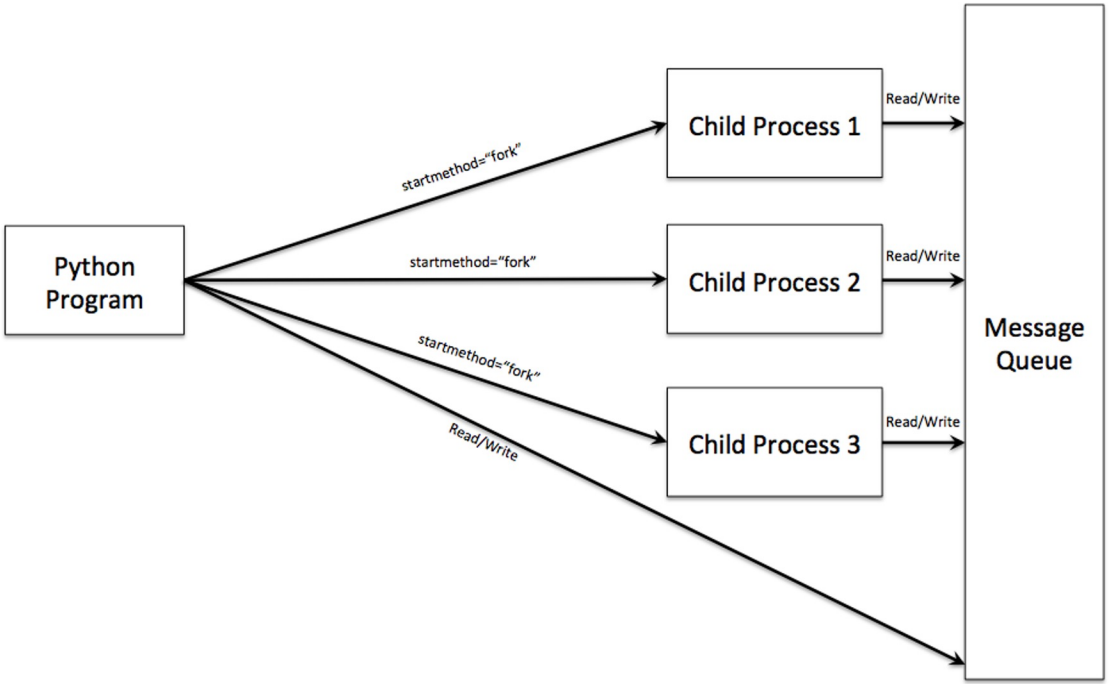
Multiprocessing in Python



Multiprocessing in Python

2. `fork` (disponibile solo su Unix, dove è il metodo di default)
    - Il processo padre **utilizza la system call fork()** per avviare un nuovo *Python interpreter process*
    - Il processo **figlio** all'avvio **eredita tutte le risorse del processo padre**
    - In questo caso la fork() viene utilizzata su un programma multithread, il che non è garantito sia safe

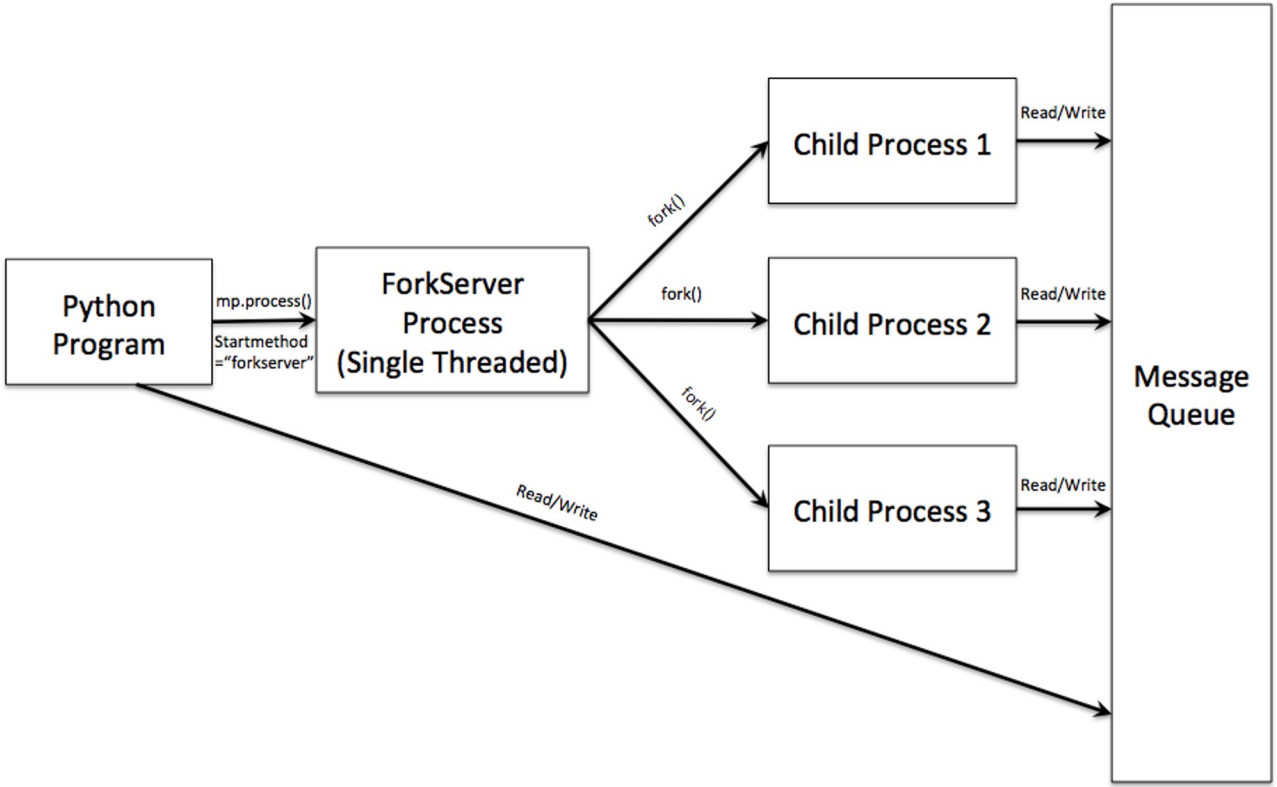
Multiprocessing in Python



Multiprocessing in Python

3. **`forkserver`** (disponibile sui sistemi Unix che supportano il passaggio di file descriptor attraverso Unix *pipe*)
    - **Viene creato un *server process***
    - Quando un nuovo processo è necessario, il **processo padre si connette al server** richiedendo ad esso la *fork* di un nuovo processo
    - **Le risorse non necessarie non sono ereditate**
    - **Il server process è single threaded**, pertanto l'utilizzo della fork è safe.

Multiprocessing in Python



Il modulo `multiprocess`

- **`multiprocess`** è un fork del modulo multiprocessing
- Estende `multiprocessing` per fornire un meccanismo di serializzazione migliorata, utilizzando **dill**.
- `multiprocess` sfrutta il modulo `multiprocessing` per **supportare lo spawning dei processi utilizzando le API del modulo threading** della libreria standard di Python
- **Permette il trasferimento di oggetti** tra processi tramite pipe o code multi-produttore/multi-consumatore
- **Permette la condivisione di oggetti** tra processi utilizzando un *processo server* o (per dati semplici) *shared memory*

`Process` class

Un oggetto della classe **Process** rappresenta una attività da eseguire in un processo separato.

La classe Process possiede metodi equivalenti a quelli della classe *threading.Thread*.

- **`Process(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)`**: Costruttore della classe Process
- **`start()`**: Avvia il processo, portando all'esecuzione del metodo *run* non appena il processo sarà schedulato
- **`run()`**: Metodo che contiene il corpo del processo, e che ne rappresenta il comportamento
- **`join(timeout=None)`**: Metodo bloccante, che attende fino alla terminazione (naturale o con eccezioni) del processo
- **`is_alive()`**: Metodo che ritorna se il Thread è alive o meno

`Process` class

Come per la classe `Thread`, anche la classe `Process` permette la creazione di un processo in vari modi, usando **callable object** o **estendendo la classe Process** stessa

Istanziare un oggetto Process, passando un "callable object" al costruttore della classe

```
import multiprocess as mp

def func():
   print ('Process running')
   return


if __name__ == '__main__':

    # creating process
    p = mp.Process(target = func)

    # starting process
    p.start()

    # wait until the process finishes
    p.join()
```

Creare una classe che estende quella Process, ridefinendo il metodo run()

```
import multiprocess as mp

class MyProcess(mp.Process):

    def run(self):
       print ('Process running')
       return

if __name__ == '__main__':

    # creating process
    p = MyProcess()

    # starting process
    p.start()

    # wait until the process finishes
    p.join()
```

Sincronizzazione tra processi

Il modulo *multiprocessing* contiene primitive di sincronizzazione **equivalenti a quelle fornite nel modulo `threading`.**

Pertanto anche con i processi è possibile utilizzare *Lock*, *RLock*, *Semaphore*, *Condition* ed *Event*

```
from multiprocess import Process, Lock

def f(l, i):

    l.acquire()

    try:
        print('hello world', i)

    finally:

        l.release()

if __name__ == '__main__':

    lock = Lock()

    for num in range(10):

        Process(target=f, args=(lock, num)).start()
```

Scambio di oggetti tra processi

- Quando sono utilizzati processi multipli, è spesso necessario **prevedere dei canali di comunicazione** tra gli stessi che permettano di evitare l'uso di primitive di sincronizzazione.
- Il modulo multiprocessing supporta **due canali di comunicazione tra processi**:
  - ***Pipe***, che consente la connessione fra più processi
  - ***Queue****,* che supporta (di fatto implementa) il problema produttori-consumatori multipli
- Una **Pipe** è caratterizzata da una coppia di ***Connection* objects**, che rappresentano gli endpoint della pipe
- Una **Queue** è una *process shared queue*, implementata attraverso una pipe e locks/semaphores
- Sia *Pipe* che *Queue* sono *thread/process-safe*

Pipe

La funzione **`Pipe([duplex])`** messa a disposizione da multiprocessing ritorna una coppia di oggetti *Connection `(conn1, conn2)`*, interconnessi attraverso una *pipe*:

- **`duplex`:**
  - se pari a True (default) la *pipe* creata è bidirezionale (duplex)
  - se pari a False, la *pipe* sarà unidirezionale:
    - `conn1` può solo ricevere messaggi
    - `conn2` può solo inviare messaggi

I due oggetti *Connection* rappresentano i due endpoint della pipe, sui quali è possibile utilizzare metodi come:

- **send(obj):** per inviare un oggetto verso l'altro endpoint della connessione, dove sarà utilizzata una *recv()*
- **recv():** ritorna un oggetto inviato dall'altro endpoint della connessione.
  Se non vi sono ancora oggetti da ricevere, il metodo è bloccante

NOTA:

- E' importante notare che i dati nella pipe possono essere corrotti se due processi utilizzano lo stesso endpoint per leggere o scrivere dati

- Nessun rischio invece nel caso di processi che utilizzano endpoint diversi di una pipe.

`Pipe`: un esempio

```
from multiprocess import Process, Pipe

def parentData(parent):

    parent.send(['Hello'])
    parent.close()

def childData(child):

    child.send(['Bye'])
    child.close()

if __name__ == '__main__':

    parent_conn, child_conn = Pipe()

    p1 = Process(target=parentData, args=(parent_conn,))
    p1.start()

    p2 = Process(target=childData, args=(child_conn,))
    p2.start()

    print(parent_conn.recv())
    print(child_conn.recv())

    p1.join()
    p2.join()
```

Queue

La classe **Queue** fornisce una ***process shared queue*** implementata attraverso una pipe e locks/semaphores.

Quando un processo immette un elemento nella Queue, un thread è avviato e trasferisce gli oggetti da un buffer verso la pipe che implementa la coda.

- **Queue([maxsize]):** costruttore della classe che ritorna una queue di tipo FIFO
  - **maxsize:** intero che definisce il limite superiore sul numero di elementi che la coda può mantenere
    - Quando la maxsize è raggiunta, l'inserimento di un nuovo elemento è bloccato fino a quando non avviene un prelievo.
    - Se maxsize è settata a zero o ad un valore negativo, non è definito un limite superiore per la coda
- **qsize():** ritorna la dimensione della coda
- **empty():** ritorna `True` se la coda è vuota, `False` altrimenti
- **full():** ritorna `True` se la cosa è piena, `False` altrimenti

Queue

- **put(item, block=True, timeout=None):** immette l'elemento nella coda
  - **block:** booleano (default `True`) che definisce se la chiamata è bloccate o meno, nel caso in cui non ci sia spazio disponibile
  - **timeout:** identifica il massimo tempo di attesa (in secondi - default None) sulla put. Allo scadere del timeout viene sollevata una *Full exception*
- **put_nowait(item):** equivalente a *put(item, block=False)*
- **get(block=True, timeout=None):** rimuove un elemento dalla coda
  - **block:** booleano (default `True`) che definisce se la chiamata è bloccate o meno, nel caso in cui non ci siano elementi disponibili
  - **timeout:** identifica il massimo tempo di attesa (in secondi - default None) sulla get.
    Allo scadere del timeout viene sollevata una *Empty exception*
- **get_nowait():** equivalente a *get(False)*.

Esistono anche ulteriori tipologie di Queue offerte da multiprocessing: *SimpleQueue* e *JoinableQueue*

Queue: un esempio

```
from multiprocess import Process, Queue


def f(q):
    q.put([42, None, 'hello'])


if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Dati condivisi tra processi: *Shared Memory*

- Le **Shared Memory** sono uno degli strumenti offerti dal modulo `multiprocessing` per la **condivisione di dati tra processi.**

- In Python una *shared memory* è rappresentata da oggetti di tipo ***Value*** o ***Array***, che sono *process-* e *thread-safe*

- Le classi **`Value`** e **`Array`** sfruttano il modulo **`ctypes`** che è una libreria di funzioni per Python che fornisce **tipi di dato compatibili con il C**
  - Consente di chiamare funzioni in *DLL* (Windows) o shared library (Unix) e può essere usata per fare il *wrap* di queste librerie scrivendo codice Python puro

Tipi in ctype

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_time_t	time_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char* (NUL terminated)	bytes object or None
c_wchar_p	wchar_t* (NUL terminated)	string or None
c_void_p	void*	int or None

Dati condivisi tra processi: Shared Memory

`Value(typecode_or_type, *args, lock=True)`

- ritorna un *ctypes object* allocato in *shared memory*. Di default il valore di ritorno è un wrapper synchronized dell'oggetto
- **`typecode_or_type`:** determina il tipo di oggetto ritornato
- **`*args`:** rappresenta gli argomenti che saranno passati al costruttore del tipo
- **`lock`:**
  - se settato a `True` (default), verrà creato un RLock per sincronizzare gli accessi all'oggetto.
  - se è popolato con un oggetto Lock o RLock, l'oggetto sarà usato per la sincronizzazione degli accessi
  - se settato a `False` gli accessi all'oggetto non saranno protetti da un lock.

Dati condivisi tra processi: Shared Memory

`Array(typecode_or_type, size_or_initializer, *, lock=True)`

- ritorna un *ctypes object* allocato dalla *shared memory*. Di default il valore di ritorno è un wrapper synchronized dell'oggetto

- **`typecode_or_type`:** determina il tipo degli elementi dell'array che viene ritornato: può essere un ctype o un *character typecode* (e.g., 'f' per indicare float) del tipo utilizzato dal modulo `array`

- **`size_or_initializer`**:
  - intero che determina la lunghezza dell'array, il quale sarà inizializzato con zeri.
  - sequenza di valori utilizzata per inizializzare l'array; la lunghezza sarà determinata dal numero di valori previsti

- **`lock`:**
  - se settato a `True` (default), verrà creato un RLock per sincronizzare gli accessi all'oggetto.
  - se è popolato con un oggetto Lock o RLock, l'oggetto sarà usato per la sincronizzazione degli accessi
  - se settato a `False` gli accessi all'oggetto non saranno protetti da un lock.

Shared Memory. Esempio

```
from multiprocess import Process, Value, Array

def f(n, a):
    n.value = 3.1415927

    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':

    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()

    p.join()

    print(num.value)
    print(arr[:])
```

L'esempio stamperà:

```
3.1415927

[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Da notare che gli argomenti `'d'` e `'i'` usati per la creazione degli oggetti Value ed Array, rispettivamente, rappresentano:

- 'd': double precision float
- 'i': signed integer