Thread-uri in Python

Ce este un thread?

Orice sistem de operare modern poate sa dea impresia ca ruleaza mai multe programe in acelasi timp. Orice program in executie este numit *proces* in terminologia UNIX sau *task* pe Windows.

Thread-urile (fire de executie) difera de procese prin faptul ca toate variabilele globale ale procesului parinte pot fi accesate de catre threaduri si pot servi ca mediu de comunicatie. Fiecare thread are totusi propriul set de variabile locale. Din acest motiv threadurile sunt numite si *lightweight processes*.

Folosirea firelor de executie in Python

Cel mai usor mod de folosirea a thread-urilor in Python este prin modulul **threading.py**

Exemplu de creare a unui thread:

```
#!/usr/bin/env python

#simple code which uses threads

import time

from threading import Thread

class MyThread(Thread):

	def __init__(self,bignum):

		Thread.__init__(self)

		self.bignum=bignum

	def run(self):

		for l in range(10):

			for k in range(self.bignum):

				res=0

				for i in range(self.bignum):
```

```
                                res+=1

def test():

      bignum=1000

      thr1=MyThread(bignum)

      thr1.start()

      thr1.join()

if __name__=="__main__":

      test()
```

Este important de stiut faptul ca un thread nu isi incepe executia decat dupa apelarea metodei start() iar functia join() asteapta terminarea executiei.

Elemente de sincronizare

Lock

Pentru a asigura accesul exclusiv la o sectiune de cod in Python se folosesc obiecte de tip Lock. Un lock se poate afla intr-unul din doua stari: blocat sau neblocat(este creat neblocat).

Cand este neblocat si se apeleaza functia acquire() se trece in starea blocat si apelul se intoarce imediat. Cand este blocat si se apeleaza acquire(), apelul nu se intoarce decat atunci cand alt thread il deblocheaza. Deblocarea este facuta de functia release() care are rolul de a trece un obiect de tip Lock din starea blocat in neblocat.

Un exemplu de folosire a unui lock:

```
#!/usr/bin/env python

import time

from threading import Thread

from threading import Lock

class MyThread(Thread):
```

```
        def __init__(self,name,sleeptime):
                Thread.__init__(self)
                self.name=name
                self.sleeptime=sleeptime
        def run(self):
                #entering critical section
                lock.acquire()
                print self.name," Now Sleeping after Lock acquired for
        ",self.sleeptime
                time.sleep(self.sleeptime)
                print self.name," Now releasing lock and then sleeping again"
                lock.release()
                #exiting critical section
                time.sleep(self.sleeptime)# why?
def test():
        sleeptime=2
        thr1=MyThread("Thread 1:",sleeptime)
        thr2=MyThread("Thread 2:",sleeptime)
        thr1.start()
        thr2.start()
        thr1.join()
        thr2.join()
if __name__=="__main__":
        lock=Lock()
        test()
```

Event-uri

Evenimentele reprezinta una din cele mai simple metode de comunicatie intre thread-uri: un thread semnalizeaza un eveniment, iar altul asteapta ca evenimentul sa se intample. In Python, un obiect de tip event are un flag intern, initial setat pe false. Acesta poate fi setat pe true cu functia set() si resetat folosind clear(). Pentru a verifica starea flag-ului, se apeleaza functia isSet().

Un alt thread poate folosi metoda wait([timeout]) pentru a astepta ca un eveniment sa se intample (ca flag-ul sa devina true): daca in momentul apelarii wait(), flag-ul este true, thread-ul apelant nu se blocheaza, dar daca este false se blocheaza pana la setarea eventului. De altfel, la un set(), toate thread-urile care asteptau event-ul cu wait() vor fi trezite.

Conditii

O variabila de tip condition este asociata cu un lock; acesta poate fi pasat (atunci cand mai multe variabile condition partajeaza un lock) sau poate fi creat implicit. Sunt prezente aici metodele acquire() si release() care le apeleaza pe cele corespunzatoare lock-ului si mai exista functiile wait(), notify() si notifyAll(), apelabile doar daca s-a reusit obtinerea lock-ului.

Metoda wait() elibereaza lock-ul si se blocheaza in asteptarea unei notificari ca urmare a unui apel notify(), care deblocheaza un singur thread care astepta si notifyAll(), care deblocheaza toate thread-urile care asteptau conditia. De mentionat ca apelurile notify() si notifyAll() nu elibereaza lock-ul, deci un thread nu va fi trezit imediat ci doar cand cele doua apeluri de mai sus au terminat de folosit lock-ul si l-au eliberat.

Semafoare

Semafoarele sunt obiecte de sincronizare diferite de lock-uri prin faptul ca salveaza numarul de operatii de debocare efectuate asupra lor. Un semafor gestioneaza un contor intern care este decrementat de un apel acquire() si incrementat de apelul release(). Contorul nu poate ajunge la valori negative deci atunci cand este apelata functia acquire() si contorul este 0 thredul se blocheaza pana cand alt thread apeleaza release(). Atunci cand este creat un semafor contorul are valoarea 1.

```
import threading

import time

import random

# vor fi maxim 3 thread-uri active la un moment dat

maxconnections = 3
```

```
semafor = threading.Semaphore(value=maxconnections)

def folosire(x):

      global semafor

      semafor.acquire()

      print "thread ",x," : enter"

      time.sleep(random.random()*5)

      print "thread ",x," : exit"

      semafor.release()

threads = 10

threadlist = []

random.seed()

print "starting threads"

for i in range(10):

      thread = threading.Thread(target=folosire, args=(i,))

      thread.start()

      threadlist.append(thread)

for i in range(len(threadlist)):

      threadlist[i].join()

print "program finished"
```

Probleme

1) Producatori si consumatori

Problema: Fie mai multi producatori si mai multi consumatori care comunica printr-un buffer partajat, limitat la un numar fix de valori. Un producator pune cate o valoare in buffer iar un consumator poate sa ia cate o valoare din buffer.

2) Problema filozofilor

Problema: Se considera mai multi filozofi ce stau in jurul unei mese rotunde. Ei isi petrec timpul gandind sau mancand. In mijlocul mesei este o farfurie cu spaghete. Pentru a putea manca, un filozof are nevoie de doua furculite. Pe masa exista cate o furculita intre fiecare doi filozofi vecini.

Regula este ca fiecare filozof poate folosi furculitele din imediata sa apropiere. Problema este de a scrie un program care simuleaza comportarea filozofilor (Trebuie evitata situatia in care nici un filozof nu poate acapara ambele furculite).

Documentatii

Un tutorial care trebuie parcurs il gasiti aici sau daca aveti instalata si documentatia, de obicei in /usr/share/doc/python-....

Mai multe informatii despre elementele de sincronizare gasiti intr-un tutorial de pe Linux Gazette

Cand incepeti sa scrieti programe Python este bine sa aveti la indemana documentatia ce include "Tutorial"-ul, "Global module index", "Library Reference" si "Language Reference".

Nu uitati sa testati exemplele din folderul de resurse al laboratorului