UNIUNEA EUROPEANĂ

GUVERNUL ROMÂNIEI

Instrumente Structurale
2007-2013

Platformă de e-learning și curriculă e-content pentru învățământul superior tehnic

Arhitectura sistemelor de calcul

29. Threaduri si modalitati de accesare a datelor partajate intre threaduri

Elemente de sincronizare

Lock

Pentru a asigura accesul exclusiv la o sectiune de cod (cu alte cuvinte definirea unei *zone critice*) in Python se folosesc obiecte de tip `Lock`.

Un lock se poate afla intr-unul din doua stari:

- blocat
- 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 lock-ul este deblocat sau 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:

lab2-example2.py

```
1. #!/usr/bin/env python
2.
3. #Un  exemplu simplu de utilizare Lock
4.
5. import time
6. from threading import Thread
7. from threading import Lock
8.
9. class MyThread(Thread):
10.
11.       def __init__(self,name,sleeptime):
12.           Thread.__init__(self)
13.           self.name=name
14.           self.sleeptime=sleeptime
15.
16.       def run(self):
17.           # entering critical section
18.           lock.acquire()
19.
20.           print self.name," Now Sleeping after Lock acquired for
   ",self.sleeptime
21.           time.sleep(self.sleeptime)
22.           print self.name," Now releasing lock and then sleeping
   again"
23.
24.           #exiting critical section
25.           lock.release()
26.
27.           time.sleep(self.sleeptime)# why?
28.
```

```
29.         def test():
30.             sleeptime=2
31.             thr1=MyThread("Thread 1:",sleeptime)
32.             thr2=MyThread("Thread 2:",sleeptime)
33.             thr1.start()
34.             thr2.start()
35.             thr1.join()
36.             thr2.join()
37.         test = staticmethod(test)
38.
39.     if __name__=="__main__":
40.         lock=Lock()
41.         MyThread.test()
```

Obiecte Eveniment (Events)

Evenimentele reprezinta una din cele mai simple metode de comunicatie intre (doua) 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 notificate (si eligibile deci pentru a rula).

Iata un exemplu:

lab2-example3.py

```
1. #!/usr/bin/env python
2.
3. import thread
4. import time
5. from threading import *
6.
7. def event_set(event):
8.     time.sleep(2)
9.     while 1 :
10.             #we wait for the flag to be set.
11.             while not event.isSet():
12.                 event.wait()
13.             print currentThread(),"...Woken Up"
14.             event.clear()
15.
16.     if __name__=="__main__":
17.         event=Event()
18.         thread.start_new_thread(event_set,(event,))
19.         while 1:
```

```
20.           event.set()
21.           print event," Has been set"
22.           time.sleep(2)
```

Un alt exemplu ce implementeaza o aplicatie ce executa periodic un task:

lab2-example3.py

```
1. import threading
2.
3. class TaskThread(threading.Thread):
4.     """Thread that executes a task every N seconds"""
5.
6.     def __init__(self):
7.         threading.Thread.__init__(self)
8.         self._finished = threading.Event()
9.         self._interval = 15.0
10.
11.        def setInterval(self, interval):
12.            """Set the number of seconds we sleep between
   executing our task"""
13.            self._interval = interval
14.
15.        def shutdown(self):
16.            """Stop this thread"""
17.            self._finished.set()
18.
19.        def run(self):
20.            while 1:
21.                if self._finished.isSet(): return
22.                self.task()
23.
24.                # sleep for interval or until shutdown
25.                self._finished.wait(self._interval)
26.
27.        def task(self):
28.            """The task done by this thread - override in
   subclasses"""
29.            pass
30.
31.    if __name__ == '__main__':
32.        class PrintTaskThread(TaskThread):
33.            def task(self):
34.                print 'running'
35.
36.        tt = printTaskThread()
37.        tt.setInterval(3)
38.        print 'starting'
39.        tt.start()
40.        print '[manager]: started, waiting now...'
41.        import time
42.        time.sleep(10)
43.        print 'Shutdown ....'
44.        tt.shutdown()
45.        print 'Done'
```

Conditii

O variabila de tip conditie (*Condition*) este asociata cu un obiect mutex *Lock*. Acesta poate fi transmis ca parametru 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.

Rolul metodelor este urmatorul:

- metoda `wait()` elibereaza lock-ul si se blocheaza in asteptarea unei notificari
- metoda `notify()` deblocheaza un singur thread aflat in asteptare
- metoda `notifyAll()` deblocheaza toate thread-urile care asteptau indeplinirea conditiei

De mentionat ca apelurile `notify()` si `notifyAll()` nu elibereaza lock-ul, deci un thread nu va fi trezit imediat ci doar cand apeluri de mai sus au terminat de folosit lock-ul si l-au eliberat.

Semafoare

Semafoarele sunt obiecte de sincronizare similare *Lock-urilor* insa difera de acestea prin faptul ca salveaza numarul de operatii de deblocare 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.

lab2-example4.py

```
1. #!/usr/bin/env python
2.
3. import threading
4. import time
5. import random
6.
7. # vor fi maxim 3 thread-uri active la un moment dat
8.
9. maxconnections = 3
10.    semafor = threading.Semaphore(value=maxconnections)
11.
12.    def folosire(x):
13.        global semafor
14.        semafor.acquire()
15.        print "thread ",x," : enter"
16.        time.sleep(random.random()*5)
17.        print "thread ",x," : exit"
18.        semafor.release()
```

```
19.
20.
21.   threads = 10
22.   threadlist = []
23.   random.seed()
24.   print "starting threads"
25.
26.   for i in range(10):
27.       thread = threading.Thread(target=folosire, args=(i,))
28.       thread.start()
29.       threadlist.append(thread)
30.
31.   for i in range(len(threadlist)):
32.       threadlist[i].join()
33.
34.   print "program finished"
```