Laboratorul 03 - Programare concurentă în Python (continuare)

Obiective

Vom continua în cadrul acestui laborator prezentarea elementelor de sincronizare oferite de Python.

Obiecte de sincronizare

Event

Event [http://docs.python.org/3/library/threading.html#event-objects]-urile sunt obiecte simple de sincronizare care permit mai multor thread-uri blocarea voluntară până la apariția unui eveniment semnalat de un alt thread (ex: o condiție a devenit adevărată). Intern, un obiect *Event* conține un flag setat inițial la valoarea *false*. El oferă următoarele operații:

- *set() [http://docs.python.org/3/library/threading.html#threading.Event.set]* - care setează valoarea flag-ului pe *true*
- *wait() [http://docs.python.org/3/library/threading.html#threading.Event.wait]* - care blochează execuția thread-urilor apelante până când flag-ul devine *true*. Dacă flag-ul este deja *true* în momentul apelării lui *wait()* aceasta nu va bloca execuția. În momentul setării flag-ului pe *true* toate thread-urile blocate în *wait()* vor fi deblocate și își vor continua execuția.
- *clear() [http://docs.python.org/3/library/threading.html#threading.Event.clear]* - setează flag-ul intern la valoarea *false* (resetează evenimentul)
- *is_set() [http://docs.python.org/3/library/threading.html#threading.Event.is_set]* - care oferă posibilitatea de interogare a valorii curente a flag-ului.

Testarea valorii flag-ului cu *is_set()* într-o buclă, fără a executa calcule utile în acea buclă sau fără a apela metode blocante, reprezintă o formă de *busy-waiting*. Această utilizare trebuie evitată, deoarece, ca orice *busy-waiting*, irosește timp de procesor care ar putea fi folosit de celelate thread-uri.

Refolosirea obiectelor *Event* pentru a semnaliza un eveniment (cu metoda *clear()*) trebuie făcută cu grijă. A doua setare a flag-ului pe *true* poate fi ștearsă de un *clear()* întârziat, înainte ca thread-ul care dorește să aștepte evenimentul să facă *wait()* pentru a doua oară. Acest lucru va duce cel mai probabil la deadlock.

O altă problemă care poate apărea în cazul refolosirii unui obiect *Event* este că un thread care dorește o a doua așteptare și-ar putea continua execuția, fără ca evenimentul să fie semnalizat a doua oară. Această situație apare atunci când resetarea flag-ului cu metoda *clear()* este întârziată mai mult decât al doilea *wait()*. Evenimentul va rămâne astfel la valoarea *true*, iar al doilea *wait()* nu va bloca execuția în acestă situație precum se dorește.

Capabilitatea obiectului *Event* de a bloca execuția thread-urilor și de a le debloca pe toate în același timp poate fi folosită în locul semaforului, ca **mecanism de blocare/deblocare**, la implementarea unei bariere ne-reentrante. Avantajul acestei soluții față de cea cu semafor este claritatea. *Event*-ul se ocupă de blocarea/deblocarea thread-urilor, iar contorul și *lock*-ul țin evidența thread-urilor intrate în barieră. Soluția cu semafor conține însă două contoare (unul dat de semaforul însuși), care trebuie să rămână corelate, iar **mecanismul de blocare/deblocare** este combinat cu **contorul**, complicând astfel analiza implementării.

simple_barrier_event.py

```
from threading import *

class SimpleBarrier():
```

```
    def __init__(self, num_threads):
        self.num_threads = num_threads
        self.count_threads = self.num_threads    # contorizeaza numarul de thread-uri ramase
        self.count_lock = Lock()                 # protejeaza accesarea/modificarea contorului
        self.threads_event = Event()             # blocheaza thread-urile ajunse

    def wait(self):
        with self.count_lock:
            self.count_threads -= 1
            if self.count_threads == 0:          # a ajuns la bariera si ultimul thread
                self.threads_event.set()         # deblocheaza toate thread-urile
        self.threads_event.wait()                # num_threads-1 threaduri se blocheaza aici
                                                 # ultimul thread nu se va bloca deoarece event-ul a fost setat

class MyThread(Thread):
    def __init__(self, tid, barrier):
        Thread.__init__(self)
        self.tid = tid
        self.barrier = barrier

    def run(self):
        print ("I'm Thread " + str(self.tid) + " before\n")
        self.barrier.wait()
        print ("I'm Thread " + str(self.tid) + " after barrier\n")
```

Bariera obținută cu un singur obiect *Event* este însă ne-reentrantă. Încercări de transformare a acestei implementări într-o barieră reentrantă, prin resetarea evenimentului cu metoda *clear()*, vor duce fie la deadlock, fie la o barieră reentrantă care nu funcționează corect în momentul întârzierii apelului *clear()* (problemele care pot apărea la reutilizarea a unui obiect *Event* sunt exemplificate mai sus). O barieră reentrantă poate fi însă ușor implementată cu două obiecte *Event*, asemănător cu folosirea a două semafoare.

Condition

Condition [http://docs.python.org/3/library/threading.html#condition-objects] (sau variabilă condiție) este un obiect de sincronizare care permite mai multor thread-uri blocarea voluntară până la apariția unei condiții semnalate de un alt thread, asemenător *Event*-urilor. Spre deosebire de acestea însă, un obiect *Condition* oferă un set de operații diferit și este asociat întotdeauna cu un *lock*. Lock-ul este creat implicit la instanțierea obiectului *Condition* sau poate fi pasat prin intermediul constructorului dacă mai multe obiecte *Condition* trebuie să partajeze același lock.

Un obiect *Condition* oferă operațiile:

- *acquire() [http://docs.python.org/3/library/threading.html#threading.Condition.acquire]* - blochează lock-ul
- *release() [http://docs.python.org/3/library/threading.html#threading.Condition.release]* - eliberează lock-ul
- *wait() [http://docs.python.org/3/library/threading.html#threading.Condition.wait]*
    - va bloca thread-ul apelant până la semnalizarea condiției prin notify de către alt thread
    - apelul wait realizează următorii pași în mod atomic:
        - eliberează lock-ul
        - așteaptă
        - blochează lock-ul atunci când thread-ul e trezit
- *notify() [http://docs.python.org/3/library/threading.html#threading.Condition.notify]*, *notify_all() [http://docs.python.org/3/library/threading.html#threading.Condition.notify_all]* - deblochează un singur thread, respectiv pe toate.

 Operațiile wait(), notify() și notify_all() trebuie întotdeauna apelate doar după blocarea prealabilă a lock-ului asociat.

Cele trei operații: *wait()*, *notify()* și *notify_all()* vor lăsa lock-ul asociat în starea blocat, deblocarea acestuia făcându-se manual cu metoda *release()*. De remarcat că după un *notify()* sau *notify_all()*, thread-urile

blocate în *wait()* nu vor continua imediat, ele trebuind să aștepte până când lock-ul asociat devine și el disponibil.

Funcționarea unui *Condition* este asemănătoare cu a monitorului asociat fiecărui obiect Java, metodele *wait()*, *notify()* și *notify_all() / notifyAll()* având aceeași semantică în ambele limbaje. Apelarea metodelor *acquire()* și *release()* este înlocuită în Java de blocul *synchronize*. Pentru o asemănare mai mare cu Java, în Python puteți folosi instrucțiunea *with* pentru apelarea automată a metodelor *acquire()* și *release()*, precum în exemplul următor:

Java	Python

`synchronize(c) {`
`  while(!check())`
`  c.wait();`
`}`
	`with c:`
	`  while(not check()):`
	`  c.wait()`

Un obiect *Condition* este util atunci când pe lângă semnalizarea unei condiții este necesar și un lock pentru a sincroniza accesul la o resursă partajată. În acest caz, un obiect *Condition* este de preferat unui *Event* deoarece oferă acest lock în mod implicit, revenirea din *wait()* în momentul semnalizării condiției făcându-se cu lock-ul blocat.

Queue

Cozile sincronizate sunt implementate în Python în modulul Queue [http://docs.python.org/3/library/queue.html] în clasele *Queue*, *LifoQueue* și *PriorityQueue*. Obiectele de aceste tipuri sunt folosite pentru implementarea comunicării între threaduri, după modelul producători-consumatori.

Metodele oferite de aceste clase permit adăugarea și scoaterea de elemente într-un mod sincronizat (*put(item) [https://docs.python.org/3/library/queue.html#Queue.Queue.put]* și *get() [https://docs.python.org/3/library/queue.html#Queue.Queue.get]*) și interogarea stării cozii (*empty() [https://docs.python.org/3/library/queue.html#Queue.Queue.empty]*, *qsize() [https://docs.python.org/3/library/queue.html#Queue.Queue.qsize]* și *full() [https://docs.python.org/3/library/queue.html#Queue.Queue.full]*). În plus față de acestea, putem implementa ușor modelul master-worker folosind metodele *task_done() [https://docs.python.org/3/library/queue.html#Queue.Queue.task_done]* și *join() [https://docs.python.org/3/library/queue.html#Queue.Queue.join]*, ca în exemplul [https://docs.python.org/3/library/queue.html#Queue.Queue.join] din documentație.

ThreadPoolExecutor

Python oferă începând cu versiunea 3.2 implementări pentru Executors și pool-uri de thread-uri sau de procese, în modulul concurrent.futures [https://docs.python.org/3/library/concurrent.futures.html?]. Un alt modul care oferă obiecte pentru pool-uri este multiprocessing [https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing] pentru pool-uri de procese și multiprocessing.dummy [https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing.dummy] pentru pool-uri de threaduri. Pentru lucrul asincron pe thread-uri recomandăm însă obiectele din concurrent.futures.

Ca și în alte limbaje (e.g. Java [https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/Executor.html]) folosirea unui Executor sau thread pool este modalitatea recomandată pentru lucrul asincron pe thread-uri. Avantajul principal este că elimină din overhead-ul creării și distrugerii de noi threaduri. Atunci când aveți de rulat un task asincron, de exemplu o operație I/O sau un call către un server, în loc să vă creați propriul thread, submiteți unui Executor funcția respectivă (un callable).

Exemple cod pentru ThreadPoolExecutor găsiți chiar în documentație [https://docs.python.org/3/library/concurrent.futures.html#threadpoolexecutor]

Pentru a submite joburi către ThreadPoolExecutor se folosesc metodele moștenite din clasa **Executor**:

- submit [https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.Executor.submit] - dă spre execuție un callable
- map [https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.Executor.map] - aplică o funcție pe o structură de date iterabilă, cum se poate observa în exemplul următor

threadpool_example.py

```
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed
from threading import current_thread
import time, random

data = ["lab1", "lab2", "lab3"]

def modify_msg(msg):
    time.sleep(random.randint(1,5))
    return "Completed: [" + msg.title() + "] in thread " + str(current_thread())

def main():
   with ThreadPoolExecutor(max_workers = 2) as executor:
      results = executor.map(modify_msg, data)

   for result in results:
      print(result)

if __name__ == '__main__':
   main()
```

Outputul programului va fi

```
Completed: [Lab1] in thread <Thread(ThreadPoolExecutor-0_0, started daemon 123145477799936)>
Completed: [Lab2] in thread <Thread(ThreadPoolExecutor-0_1, started daemon 123145494589440)>
Completed: [Lab3] in thread <Thread(ThreadPoolExecutor-0_0, started daemon 123145477799936)>
```

Studiu de caz - Bariera

De multe ori este necesar ca un grup de thread-uri să ajungă toate într-un anumit punct al execuției (ex: fiecare thread a calculat un rezultat intermediar al algoritmului) și numai după aceea să își continue execuția (ex: rezultatele intermediare sunt partajate de toate thread-urile în partea următoare a algoritmului). Mecanismul de sincronizare potrivit pentru asemenea situații este **bariera**.

Începând cu Python 3.2 în modulul *threading* a fost introdusă clasa *Barrier [https://docs.python.org/3/library/threading.html#barrier-objects]*, acesta fiind o barieră reentrantă implementată folosind variabile condiție (cod sursă [https://github.com/python/cpython/blob/master/Lib/threading.py#L580]). În această secțiune vom prezenta două variante pentru implementarea unui astfel de obiect.

Ce trebuie să ofere o barieră?

- un **mecanism de blocare** a thread-urilor - Barrier#wait() [https://docs.python.org/3/library/threading.html#threading.Barrier.wait]
- un **contor** al numărului de thread-uri care au ajuns/mai trebuie să ajungă la barieră - Barrier#n_waiting [https://docs.python.org/3/library/threading.html#threading.Barrier.n_waiting]
- **deblocarea** tuturor thread-urilor atunci când a ajuns și ultimul dintre ele la barieră

Barieră - varianta semafor

Bariera ne-reentrantă

Putem implementa o barieră folosind un semafor inițializat cu 0 (**mecanismul de blocare/deblocare**) și un contor al numărului de thread-uri care mai trebuie să ajungă la barieră (**contorul**), inițializat cu numărul de thread-uri utilizate. Semaforul este folosit pentru a bloca execuția thread-urilor. Contorul este decrementat de fiecare thread care ajunge la barieră și reprezintă numărul de thread-uri care au mai rămas de ajuns. Fiind o variabilă partajată, modificarea lui trebuie bineînțeles protejată de un *lock*. În momentul în care ultimul thread decrementează contorul, acesta va avea valoarea 0, semnalizând faptul că toate thread-urile au ajuns la barieră. Ultimul thread va incrementa astfel semaforul (**deblocarea**) și va debloca toate thread-urile blocate.

simple_barrier_semaphore.py

```
from threading import *

class SimpleBarrier():
    def __init__(self, num_threads):
        self.num_threads = num_threads
        self.count_threads = self.num_threads    # contorizeaza numarul de thread-uri ramase
        self.count_lock = Lock()                 # protejeaza accesarea/modificarea contorului
        self.threads_sem = Semaphore(0)          # blocheaza thread-urile ajunse

    def wait(self):
        with self.count_lock:
            self.count_threads -= 1
            if self.count_threads == 0:          # a ajuns la bariera si ultimul thread
                for i in range(self.num_threads):
                    self.threads_sem.release()   # incrementarea semaforului va debloca num_threads thread-uri
        self.threads_sem.acquire()               # num_threads-1 threaduri se blocheaza aici
                                                 # contorul semaforului se decrementeaza de num_threads ori

class MyThread(Thread):
    def __init__(self, tid, barrier):
        Thread.__init__(self)
        self.tid = tid
        self.barrier = barrier

    def run(self):
        print ("I'm Thread " + str(self.tid) + " before\n")
        self.barrier.wait()
        print ("I'm Thread " + str(self.tid) + " after barrier\n")
```

De ce nu este reentrantă bariera cu un semafor?

Fie cazul în care avem N thread-uri, iar acestea trebuie sincronizate prin barieră de mai multe ori:

- *N-1* thread-uri vor face *acquire* pe semafor
- ultimul thread face *release* de *N* de ori pe semafor
    - unul din *release*-uri este pentru el
- *N-1* thread-uri se deblochează și își continuă execuția; ultimul thread ar trebui să facă *acquire* și să nu se blocheze, doarece semaforul ar trebui să fie 1
- rulând în buclă însă, unul din thread-urile deblocate poate ajunge să facă *acquire* din nou, înainte ca ultimul thread să treacă de *acquire*
    - ultimul thread va rămâne blocat la *acquire*, urmând ca și celelalte thread-uri să se blocheze în al doilea acces al barierei; nici un thread nu mai face *release*, ducând astfel la deadlock

Bariera reentrantă

Barierele reentrante (eng. *reusable barrier*) sunt utile în prelucrări 'step-by-step' și/sau bucle. Unele aplicații pot necesita ca thread-urile să execute anumite operații în buclă, cu rezultatele tuturor thread-urilor din iterația curentă necesare pentru începerea iterației următoare. În acest caz, după fiecare iterație, se folosește o sincronizare cu barieră reentrantă.

Pentru a adapta bariera din secțiunea anterioară astfel încât să poată fi folosită de mai multe ori, avem nevoie de încă un semafor. Soluția aceasta se bazează pe necesitatea ca toate cele N thread-uri să treacă

de *acquire()* înainte ca vreunul să revină la barieră. Astfel, partea de sincronizare este compusă din două etape, fiecare folosind câte un semafor.

Folosind implementarea de mai jos, garantăm că thread-urile ajung să se blocheze din nou pe primul semafor doar după ce **toate** au trecut în prealabil de acesta:

- *N-1* thread-uri vor face *acquire* pe semaforul 1
- ultimul thread face *release* de *N* de ori pe semaforul 1
  - unul din *release*-uri este pt el
- *N-1* thread-uri se deblochează și fac *acquire* pe semaforul 2
- ultimul thread face și el *acquire* pe semaforul 1 și trece de acesta
- ultimul thread face *release* de *N* de ori pe semaforul 2
- *N-1* thread-uri se deblochează și fac *acquire* pe semaforul 1

s.a.m.d….

reusable_barrier_semaphore.py

```
from threading import *

class ReusableBarrier():
    def __init__(self, num_threads):
        self.num_threads = num_threads
        self.count_threads1 = [self.num_threads]
        self.count_threads2 = [self.num_threads]
        self.count_lock = Lock()                # protejam accesarea/modificarea contoarelor
        self.threads_sem1 = Semaphore(0)        # blocam thread-urile in prima etapa
        self.threads_sem2 = Semaphore(0)        # blocam thread-urile in a doua etapa

    def wait(self):
        self.phase(self.count_threads1, self.threads_sem1)
        self.phase(self.count_threads2, self.threads_sem2)

    def phase(self, count_threads, threads_sem):
        with self.count_lock:
            count_threads[0] -= 1
            if count_threads[0] == 0:           # a ajuns la bariera si ultimul thread
                for i in range(self.num_threads):
                    threads_sem.release()       # incrementarea semaforului va debloca num_threads thread-uri
                count_threads[0] = self.num_threads  # reseteaza contorul
        threads_sem.acquire()                   # num_threads-1 threaduri se blocheaza aici
                                                # contorul semaforului se decrementeaza de num_threads ori

class MyThread(Thread):
    def __init__(self, tid, barrier):
        Thread.__init__(self)
        self.tid = tid
        self.barrier = barrier

    def run(self):
        for i in range(10):
            self.barrier.wait()
            print ("I'm Thread " + str(self.tid) + " after barrier, in step " + str(i) + "\n")
```

Barieră - varianta condition

O altă utilizare a obiectului *Condition* poate fi văzută în implementarea barierei reentrante, ca **mecanism de blocare/deblocare**. Bariera poate fi implementată cu un singur obiect deoarece prezența implicită a lock-ului în operațiile obiectului *Condition*, împreună cu funcționarea atomică a metodei *wait()* ne permit evitarea problemelor ce apar la refolosirea obiectelor *Event*. Pe lângă notificarea thread-urilor de îndeplinirea condiție barierei, putem folosi obiectul *Condition* și pentru protejarea resursei partajate (**contorul** de thread-uri blocate), eliminând astfel necesitatea unui *Lock* separat.

reusable_barrier_condition.py

```
from threading import *

class ReusableBarrier():
    def __init__(self, num_threads):
        self.num_threads = num_threads
        self.count_threads = self.num_threads    # contorizeaza numarul de thread-uri ramase
        self.cond = Condition()                  # blocheaza/deblocheaza thread-urile
                                                 # protejeaza modificarea contorului

    def wait(self):
        self.cond.acquire()                      # intra in regiunea critica
        self.count_threads -= 1;
        if self.count_threads == 0:
            self.cond.notify_all()               # deblocheaza toate thread-urile
            self.count_threads = self.num_threads    # reseteaza contorul
        else:
            self.cond.wait();                    # blocheaza thread-ul eliberand in acelasi timp lock-ul
        self.cond.release();                     # iese din regiunea critica


class MyThread(Thread):
    def __init__(self, tid, barrier):
        Thread.__init__(self)
        self.tid = tid
        self.barrier = barrier

    def run(self):
        for i in range(10):
            self.barrier.wait()
            print ("I'm Thread " + str(self.tid) + " after barrier, in step " + str(i) + "\n")
```

Puteți găsi aici modul de implementare a barierei cu condition [https://github.com/python/cpython/blob/master/Lib/threading.py#L580] din sursele oficiale.

Exerciții

Task 1-Condition

Pornind de la fișierul `event.py`, înlocuiți obiectele *Event* `work_available` și `result_available` cu o singură variabilă condiție, păstrand funcționalitatea programului intactă.

Task 2-Events

Rulați fișierul `broken-event.py`. Încercați diferite valori pentru *sleep()*-ul de la linia 47. Încercați eliminarea apelului *sleep()*. Ce observați? Precizați secvența _minimă_ de intercalare a apelurilor `set` și `wait` pe cele două obiecte *Event* care generează comportamentul observat.

- Folosiți `Ctrl+\` pentru a opri un program blocat.
- Folosiți *sleep()* pentru a forța diferite intercalări ale thread-urilor.
- Folosiți instrucțiuni *print* înaintea metodelor care lucrează cu `Event`-uri pentru a avea o idee asupra ordinii operațiilor.
- Datorită intercalărilor thread-urilor este posibil ca *print*-urile să nu reflecte ordinea exactă a operațiilor. Rulați de mai multe ori pentru a putea prinde problema.

Task 3-ThreadPoolExecutor

Folosind un pool de thread-uri căutați o secvență de ADN într-un set de eșantioane de ADN (DNA samples). Creați-vă un modul în care:

1. Folosiți modulul `random` pentru a genera 100 sample-uri de DNA de lungime 10000
    - random.choice(seq) [https://docs.python.org/3/library/random.html#random.choice] întoarce un element random dintr-o secvența data (în Python, tipurile secvența (sequence types) sunt și

listele și stringurile, deci puteți folosi șirul 'ATGC' ca input pt choice()).
   - folosiți seed [https://docs.python.org/3/library/random.html#random.seed] cu un parametru pentru a genera tot timpul aceleași secvențe random și a va testa codul mai ușor. By default seed-ul este luat pe baza timpului curent, deci se va schimbă între rulări.
   - "the pythonic way"-încercați să generați secvențele într-o singură linie (one liner cu list comprehensions)
2. Definiți-vă un subșir cu mai mult de 10 elemente. Recomandăm să afișați sample-urile obținute random și să selectați un subșir din una din ele.
3. Folosiți un `ThreadPoolExecutor` cu un număr maxim de threaduri, de exemplu 30
4. Funcția pe care o execută fiecare thread:
   - primește ca argument indexul unui sample
   - caută prima apariție a subșirului în sample-ul dat.
   - întoarce un mesaj, de exemplu "DNA sequence found în sample 1"
5. Folosind submit [https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.Executor.submit] sau map [https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.Executor.map] dați spre execuție funcția de căutare.
6. Afișați rezultatele
   - păstrați ce întorc apelurile către submit sau rezultatul lui map într-o variabilă și apoi iterați pe elementele din ea, afișând fiecare rezultat)
   - puteți vedea în exemplul din laborator cum se afișează rezultatele unui map, sau în documentație pentru submit [https://docs.python.org/3/library/concurrent.futures.html#threadpoolexecutor-example]
     - dacă nu sunteți familiari cu sintaxa de one-liners folosită în documentație, puteți folosi simple for-uri, va creați o lista goală înainte, și la fiecare iterație din for adăugați în ea rezultatul metodei submit. După aceea mai faceți un for în care afișați elementele din lista, apelând .result() pe fiecare din ele.

Resurse

- PDF laborator
- Schelet laborator
- Soluție laborator

Referințe

- modulul threading [https://docs.python.org/3/library/threading.html] - Thread, Lock, Semaphore, Condition, Event, Barrier
- modulul Queue [http://docs.python.org/3/library/queue.html#module-Queue]
- modulul concurrent.futures [https://docs.python.org/3/library/concurrent.futures.html#module-concurrent.futures]
- Little book of semaphores - capitolele 3.5 *Barrier* și 3.6 *Reusable Barrier*

asc/laboratoare/03.txt · Last modified: 2021/02/16 20:40 (external edit)