Programare concurentă în Python (continuare)

Obiective

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

Obiecte de sincronizare

Event

Event-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 două operații de bază: *set()*, care setează valoarea flag-ului pe *true*, și *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.

Event-ul mai oferă și alte două operații: *clear()*, care setează flag-ul intern la valoarea *false* (resetează evenimentul) și *is_set()*, care oferă posibilitatea de interogare a valorii curente a flag-ului.

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

O altă problemă care poate apărea în cazul refolosirii unui obiect *Event* este că thread-ul care dorește așteptarea își poate continua execuția fără ca evenimentul să fie semnalizat a doua oară. Această situație apare atunci când flagul nu este resetat cu metoda *clear()* înainte de al doilea *wait()*, el rămânând astfel la valoarea *true*. Al doilea *wait()* nu va bloca execuția în acestă situație precum se dorește.

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* și trebuie evitată, deoarece, ca orice

busy-waiting, irosește timp de procesor, care ar putea fi altfel folosit de celelate thread-uri.

Mai jos este prezentat un exemplu care folosește obiecte *Event* atât pentru a aștepta îndeplinirea unei condiții (*work_available* și *result_available*), cât și doar pentru a testa un flag (*terminate*). *work_available* este folosit pentru a bloca worker-ul atunci când nu are task-uri de procesat, iar *result_available* este folosit pentru a bloca masterul cât timp worker-ul își procesează task-ul. *terminate* este folosit pentru a semnaliza worker-ului că trebuie să-și încheie execuția. Observați că deși se folosește metoda *is_set()*, în buclă există alte apeluri blocante, care fac worker-ul să testeze flagul doar la apariția unui task, deci nu putem spune că avem de-a face cu *busy-waiting*.

event.py

```
from threading import enumerate, Event, Thread

class Master(Thread):
    def __init__(self, max_work, work_available, result_available):
        Thread.__init__(self, name = "Master")
        self.max_work = max_work
        self.work_available = work_available
        self.result_available = result_available

    def set_worker(self, worker):
        self.worker = worker

    def run(self):
        for i in xrange(self.max_work):
            # generate work
            self.work = i
            # notify worker
            self.work_available.set()
            # get result
            self.result_available.wait()
            self.result_available.clear()
            if self.work + 1 != self.worker.get_result():
                print "oops",
            print "%d -> %d" % (self.work, self.worker.get_result())

    def get_work(self):
        return self.work

class Worker(Thread):
    def __init__(self, terminate, work_available, result_available):
        Thread.__init__(self, name = "Worker")
        self.terminate = terminate
        self.work_available = work_available
        self.result_available = result_available

    def set_master(self, master):
        self.master = master

    def run(self):
```

```
        while(True):
            # wait work
            self.work_available.wait()
            self.work_available.clear()
            if(terminate.is_set()): break
            # generate result
            self.result = self.master.get_work() + 1
            # notify master
            self.result_available.set()

    def get_result(self):
        return self.result

if __name__ ==  "__main__":
    # create shared objects
    terminate = Event()
    work_available = Event()
    result_available = Event()

    # start worker and master
    w = Worker(terminate, work_available, result_available)
    m = Master(10, work_available, result_available)
    w.set_master(m)
    m.set_worker(w)
    w.start()
    m.start()

    # wait for master
    m.join()

    # wait for worker
    terminate.set()
    work_available.set()
    w.join()

    # print running threads for verification
    print enumerate()
```

Condition

Condition (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, asemănător *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()* și *release()* care vor bloca, respectiv, elibera lock-ul asociat și operațiile specifice: *wait()*, *notify()* și *notify_all()*. Aceste ultime trei operații trebuie întotdeauna apelate doar după blocarea prealabilă a lock-ului asociat. *wait()* va cauza blocarea thread-ul apelant până la semnalizarea condiției de către alt thread. Înainte de blocarea thread-ului apelant, *wait()* va debloca lock-ul asociat, iar după semnalizarea condiției, metoda *wait()* va aștepta preluarea lock-ului înainte de terminare. Toți acești trei pași ai metodei *wait()* sunt efectuați în mod atomic, thread-ul apelant fiind lăsat în aceeași stare ca înainte de apel (cu lock-ul asociat blocat). Semnalizarea unei condiții se face cu metodele *notify()* sau *notify_all()*. Diferența dintre aceste două metode este numărul de thread-uri deblocate în momentul apelului: *notify()* va debloca un singur thread, iar *notify_all()* va debloca toate thread-urile.

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.

Un obiect *Condition* este folosit 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.

Un exemplu de folosire a obiectelor *Condition* poate fi găsit în secțiunea *Bariere* unde, pe lângă notificarea thread-urilor de îndeplinirea unei condiții, avem și o resursă partajată (contorul de thread-uri blocate) care trebuie protejată de un lock.

Queue

Cozile sincronizate sunt implementate în Python în modulul Queue în clasele *Queue*, *LifoQueue* și *PriorityQueue*. Obiectele de acest 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* și *get*, și interogarea stării cozii, *empty*, *qsize* și *full*. În plus față de acestea, putem implementa comportamentul producător-consumator sau master-worker folosind metodele *task_done* și *join*, ca în exemplul din documentație.

Bariere

De multe ori un grup de thread-uri sau procese trebuie să ajungă toate într-un anumit punct și numai după aceea execuția poate continua. Mecanismul de sincronizare potrivit pentru asemenea cazuri este **bariera**. In Python 2.7 nu avem un obiect de sincronizare care să implementeze comportamentul unei bariere, dar putem s-o implementam cu ajutorul celorlalte mecanisme de sincronizare. Începând cu versiunea Python 3.2 s-a introdus clasa *Barrier* în modulul *threading*, acesta fiind o barieră reentrantă implementată folosind variabile condiție (cod sursă).

Ce trebuie să ofere o barieră?

- un mecanism prin care thread-urile se blochează când o folosesc
- trebuie să mențină câte thread-uri trebuie așteptate
- deblocarea tuturor thread-urilor atunci când a ajuns și ultimul dintre ele la barieră

Bariera ne-reentrantă

Putem implementa o barieră folosind un semafor inițializat cu 0, la care se blochează thread-urile, acestea deblocandu-se când ultimul thread ajunge la barieră și incrementează semaforul.

simple-barrier.py

```
from threading import *
class SimpleBarrier():

    def __init__(self, num_threads):
        self.num_threads = num_threads
        self.count_threads = self.num_threads
        self.counter_lock = Lock()      # protejam decrementarea
numarului de threaduri
        self.threads_sem = Semaphore(0) # contorizam numarul de
threaduri

    def wait(self):
        with self.counter_lock:
            self.num_threads -= 1
            if self.num_threads == 0:   # a ajuns la bariera si ultimul
thread
                for i in range(self.count_threads):
                    self.threads_sem.release()   # contorul semaforului
devine count_threads
        self.threads_sem.acquire()     # n-1 threaduri se blocheaza
aici
                                       # contorul semaforului se
decrementeaza de count_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 "
        barrier.wait()
        print "I'm Thread " + str(self.tid) + " after barrier"
```

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 fac *acquire*
- ultimul thread face *release* de *N* de ori
  - unul din *release*-uri este pt el
- *N-1* thread-uri se deblochează, ultimul thread ar trebui să facă *acquire* și să nu se blocheze (semaforul fiind 1).
- rulând în buclă, unul din thread-urile deblocate poate ajunge sa facă *acquire* din nou, înainte ca ultimul thread să treacă de *acquire*.
  - ultimul thread rămâne blocat la *acquire*

Bariera reentrantă

Barierele reentrante (eng. *reusable barrier*) pot fi folosite în prelucrări 'step-by-step' și bucle. Thread-urile sau procesele pot executa anumite instrucțiuni în buclă, iar toate rezultatele iterației curente sunt necesare pentru începerea iterației următoare. În acest caz, după fiecare iterație, se face o sincronizare cu barieră.

Pentru a adapta bariera din secțiunea anterioară astfel încât să poată fi folosită de mai multe ori de către aceleași thread-uri, 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 și să se blocheze din nou. 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 după ce toate au trecut de acesta.

- *N-1* thread-uri vor face *acquire* pe semaforul 1
- ultimul thread face *release* de *N* de ori pe semaforul 1
- *N-1* thread-uri se deblochează și fac *acquire* pe semaforul 2
- ultimul thread face *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....

reentrant-barier.py

```
class ReusableBarrierSem():

    def __init__(self, num_threads):
        self.num_threads = num_threads
        self.count_threads1 = self.num_threads
        self.count_threads2 = self.num_threads

        self.counter_lock = Lock()       # protejam decrementarea
numarului de threaduri
        self.threads_sem1 = Semaphore(0) # contorizam numarul de
```

```
threaduri pentru prima etapa
        self.threads_sem2 = Semaphore(0) # contorizam numarul de
threaduri pentru a doua etapa

    def wait(self):
        self.phase1()
        self.phase2()

    def phase1(self):
        with self.counter_lock:
            self.count_threads1 -= 1
            if self.count_threads1 == 0:
                for i in range(self.num_threads):
                    self.threads_sem1.release()
            self.count_threads2 = self.num_threads

        self.threads_sem1.acquire()

    def phase2(self):
        with self.counter_lock:
            self.count_threads2 -= 1
            if self.count_threads2 == 0:
                for i in range(self.num_threads):
                    self.threads_sem2.release()
            self.count_threads1 = self.num_threads

        self.threads_sem2.acquire()

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):
            barrier.wait()
            print "I'm Thread " + str(self.tid) + " after barrier, in
step " + str(i)
```

Soluția de mai sus este necesară deoarece avem nevoie de un mecanism de a notifica toate thread-urile că pot reîncepe execuția. O altă modalitate de implementare este folosind un obiect *Condition* care să trezească toate thread-urile care așteaptă la barieră. Deoarece obiectele *Condition* conțin un lock, nu mai avem nevoie de alt lock pentru protejarea decrementării numărului de thread-uri.

cond-barier.py

```
class ReusableBarrierCond():
```

```
    def __init__(self, num_threads):
        self.num_threads = num_threads
        self.count_threads = self.num_threads
        self.cond = Condition(Lock())

    def wait(self):
        self.cond.acquire()       # intra in regiunea critica
        self.count_threads -= 1;
        if self.count_threads == 0:
            self.cond.notify_all() # trezeste toate thread-urile,
acestea vor putea reintra in regiunea critica dupa release
            self.count_threads=self.num_threads
        else:
            self.cond.wait();    # iese din regiunea critica, se
blocheaza, cand se deblocheaza face acquire pe lock
        self.cond.release();     # iesim din regiunea critica
```

Exerciții

1. (**2p**) Pornind de la exemplul de folosire a obiectelor `Event` din fișierul `event.py` modificați metodele *get_work* și *get_result* astfel încât fiecare metodă să afișeze numele ei, obiectul self cu care a fost apelată și thread-ul curent pe care rulează. Ce observați? Justificați.
2. (**4p**) Rulați fișierul `broken-event.py`. Încercați diferite valori pentru *sleep()*-ul de la linia 47. Încercați eliminarea apelului *sleep()*. Ce observați? Ce intercalare a thread-urilor 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.
3. (**4p**) Implementați in fișierul `gossiping.py` o propagare ciclică de tip gossiping folosind bariere.
   - Se consideră N noduri (obiecte de tip `Thread`), cu indecși 0...N-1
   - Fiecare nod ține o valoare generată random
   - Calculați valoarea minimă folosind următorul procedeu:
     - nodurile rulează în cicluri
     - într-un ciclu, fiecare nod comunică cu un subset de alte noduri pentru a obține valoarea acestora și a o compara cu a sa
       - ca subset considerați random 3 noduri din lista de noduri primită în constructor și obțineți valoarea acestora (metoda get_value)
     - după terminarea unui ciclu, fiecare nod va avea ca valoare minimul obținut în ciclul anterior
     - la finalul iterației toate nodurile vor conține valoarea minimă
   - Folosiți una din barierele reentrante din modulul `barrier` din scheletul de laborator
   - Pentru a simplifica implementarea, e folosit un număr fix de cicluri, negarantand astfel convergența tutoror nodurilor la același minim

Resurse

- Responsabilii acestui laborator: Adriana Drăghici, Dan Dragomir
- PDF laborator
- Schelet laborator

Referințe

- modulul threading - Thread, Lock, Semaphore, Condition, Event
- modulul Queue
- Little book of semaphores - capitolele 3.5 *Barrier* și 3.6 *Reusable Barrier*