UNIUNEA EUROPEANĂ

GUVERNUL ROMÂNIEI

Instrumente Structurale
2007-2013

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

Arhitectura sistemelor de calcul

27. Notiuni si concepte de baza privind programarea cu threadeduri

Fire de executie (Threads) in Python

Ce este un thread

Pe scara larga sistemele de calcul curente pot executa operatii multiple in acelasi timp. Spre exemplu un utilizator casnic poate sa utilizeze un procesor de text in timp ce, pe acelasi calculator, alte aplicatii descarca fisiere, controleaza o coada de taskuri pentru imprimanta locala sau ruleaza un fisier video. In exemplul anterior am considerat ca sistemul de operare in sine permite rularea mai multor aplicatii simultan. Dar aceasta idee se poate extinde si la nivelul unei aceleiasi aplicatii: aplicatia ce ruleaza fisierul video online trebuie simultan sa *citeasca* continutul video de pe retea, sa il *decompreseze*, sa *actualizeze* display-ul local cu aceste informatii,etc. Spunem ca aplicatiile ce ofera aceste capabilitati constituie software concurent.

Deci ce este concurenta? Concurenta este proprietatea unei logice de program de a executa **simultan** un set de task-uri. In timp ce concurenta defineste problema, **paralelismul** reprezinta o metoda de implementare a acestei paradigme de programare concurenta ce permite rularea unui set de task-uri intr-un mod care sa utilizeze procesoare multiple, core-uri multiple sau chiar mai multe masini (intr-o structura de tip grid de exemplu).

Remember: Orice program in executie este numit *proces* in terminologia UNIX sau *task* pe Windows.

Revenind la thread-uri, acestea reprezinta o metoda specifica de implementare a concurentei. *Firele de executie* reprezinta agenti creati (*spawned*) in cadrul unui program principal ce executa concurent taskuri definite de dezvoltator. Implementarea threadurilor difera de la un sistem de operare la altul dar in cele mai multe cazuri un fir de executie este parte a unui proces. Mai multe threaduri pot exista in cadrul aceluiasi proces, partajeaza resurse comune (memorie, descriptori I/O, etc). In aceasta privinta 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*.

In cadrul unui sistem mono-procesor, rularea concurenta a mai multor fire de executie se face prin metoda *partajarii timpului de executie* (*time sharing/time division/time slicing*): sistemul de operare alterneaza succesiv executia thread-urile active (perceptia este a rularii simultane insa in realitate un singur thread ruleaza la un moment dat).

In cadrul unui sistem multi-procesor sau multi-nucleu, threadurile vor rula in general cu adevarat simultan, cu fiecare procesor ruland un thread specific.
Din punct de vedere al suportului pentru programarea multithreading limbajele se impart in doua categorii:

1. limbaje cu threaduri utilizator (*green threads*) ce nu sunt vizibile sistemului de operare ci doar la nivelul unui singur proces (Gasiti vreun dezavantaj?)

2. limbaje cu threaduri native (adesea denumite si *kernel threads*) ce sunt vizibile la nivel de sistem de operare ce permite executia lor paralela.

In cadrul acestui laborator vom folosi limbajul Python ce face parte din prima categorie din clasificarea de mai sus.

Folosirea firelor de executie in Python

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

Exemplu de creare a unui thread:

lab2-example1.py

```
1. #!/usr/bin/env python
2.
3. #simple code which uses threads
4. import time
5. from threading import Thread
6. class MyThread(Thread):
7.     def __init__(self,bignum):
8.         Thread.__init__(self)
9.         self.bignum=bignum
10.        def run(self):
11.            for l in range(10):
12.                for k in range(self.bignum):
13.                    res=0
14.                    for i in range(self.bignum):
15.                        res+=1
16.        def test():
17.            bignum=1000
18.            thr1=MyThread(bignum)
19.            thr1.start()
20.            thr1.join()
21.        test = staticmethod(test)
22.
23.    if __name__=="__main__":
24.        MyThread.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 (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"
```

Probleme clasice multi-threading

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.

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).

Problema fumatorilor

Problema: *Little Book of Semaphores pag.119*

Four threads are involved: an agent and three smokers. The smokers loop forever, first waiting for ingredients, then making and smoking cigarettes. The ingredients are tobacco, paper, and matches. We assume that the agent has an infinite supply of all three ingredients, and each smoker has an infinite supply of one of the ingredients; that is, one smoker has matches, another has paper, and the third has tobacco.

The agent repeatedly chooses two different ingredients at random and makes them available to the smokers. Depending on which ingredients are chosen, the smoker with the complementary ingredient should pick up both resources and proceed. For example, if the agent puts out tobacco and paper, the smoker with the matches should pick up both ingredients, make a cigarette, and then signal the
agent.

To explain the premise, the agent represents an operating system that allocates resources, and the smokers represent applications that need resources. The problem is to make sure that if resources

are available that would allow one more applications to proceed, those applications should be woken up. Conversely, we want to avoid waking an application if it cannot proceed.

Nota: *De rezolvat in laborator*

Taskuri

Pentru taskurile de mai jos creati un numar suficient de threaduri, uneori este nevoie chiar si de 1000 de threaduri pentru a observa comportamente ciudate.

1. Hello world

Implementati un program multi threading a carui threaduri afiseaza “Hello world”. Transmiteti fiecarui thread indexul sau care va trebui afisat in momentul rularii. De ce intre doua rulari succesive indexurile afisate apar in alta ordine? Ce trebuie facut pentru ca textul afisat sa nu fie intercalat cu alt text?

2. Thread Counter

In acest task se incearca numararea tuturor threadurilor ce ruleaza. Creati o aplicatie multi threading ce va avea urmatoarea structura:

- clasa Counter va avea o metoda inc() ce va incrementa un contor
- clasa MyThread va primi ca parametru o instanta a unui Contor si ii va apela metoda inc

Folositi in cadrul metodei run() un mecanism de busy waiting pentru a asigura pornirea simultana a threadurilor.

Cerinte:

1. Afisati la sfarsit numarul de threaduri numarate.
2. Corectati programul astfel incat numarul de threaduri numarate sa fie cel corect. Monitorizati load-ul calculatorului.
3. Inlocuiti mecanismul de busy waiting cu un element de sincronizare.

3. Producer-Consumer

Implementati problema de sincronizare clasica producator-consumator. Pot exista p producatori si c consumatori, p si c configurabile.

4. Teacher's choise

Resurse

- Threaduri - http://docs.python.org/library/threading.html

- Lockuri - http://docs.python.org/library/threading.html
- Variabile conditie - http://docs.python.org/library/threading.html
- Semafoare - http://docs.python.org/library/threading.html
- Evenimente - http://docs.python.org/library/threading.html
- Introduction to semaphores - http://www.youtube.com
- Little book of semaphore - downey08semaphores.pdf
- Understanding Threading in Python - http://linuxgazette.net/107/pai.html
- Python Threads and the Global Interpreter Lock - http://jessenoller.com/2009/02/01/python-threads-and-the-global-interpreter-lock/
- Programming on Parallel Machines (Chapter 3) - parprocbook.pdf