Laboratorul 02 - Fire de execuție în Python

Obiective

Scopul acestui laborator îl reprezintă familiarizarea cu lucrul cu thread-uri și obiecte de sincronizare în Python. Pentru acestea aveți nevoie de cunoașterea elementelor de sintaxă Python prezentate în laboratorul 1 și de lucrul cu clase prezentat în laboratorul acesta.

Recapitulare laborator 1

În cadrul laboratorului de ASC folosim versiunea 3.8 a Python-ului, suportul pentru versiunea 2.x terminându-se la data de 1 Ianuarie 2020. Această versiune este incompatibilă cu Python 2.x și unele construcții sau biblioteci este posibil să nu fie suportate nici de versiunile anterioare 3.8.

Particularități de limbaj:

- **Indentarea** este obligatorie pentru a delimita blocurile de cod.
- Este dynamically și strongly typed:
  - *dynamically typed* - pentru că tipurile variabilelor nu sunt precizate explicit în cod, și acestea se poate schimba pe măsură ce atribuim valori variabilelor
  - *strongly typed* - pentru că nu se pot face conversii de tip implicite (e.g. adunare de `string` cu `int`)
  - pentru conversiile explicite între tipurile numerice, boolean și șiruri de caractere folosiți funcțiile built-in [https://docs.python.org/3.8/library/functions.html]
- Keywords:
  - `None` este echivalentul null în Java
  - `pass` este echivalentul unui bloc {} din c/java
- Tipurile de date cele mai folosite sunt *int, float, string, boolean, list, tuple, dict*.

Un fișier de cod Python este considerat un **modul**. Pentru a folosi alte module utilizăm `import` în următoarele modalități:

Funcțiile se declară folosind keyword-ul `def` și nu li se specifică tip de return sau tipuri pentru parametri. Se poate simula supraîncărcarea (overloading) metodelor folosind parametri cu valori implicite. Funcțiile sunt de fapt obiecte.

Construcția **`if __name__ == "__main__"`** delimitează 'main'-ul unui modul. Aceasta nu este obligatorie, însă dacă nu e folosită, orice cod cu indentare top level s-ar executa de fiecare dată când fișierul este parsat (ex: când este importat).

func_example.py

```
def f(a, b="world", c=0):
    print (" ".join([a, b, str(c)]))

f("hello")                  # hello world 0
f("hello", "lab")           # hello lab 0
f("hello", "lab", 2)        # hello lab 2
f("hello", c=2)             # hello world 2
f("hello", "lab", c=2)      # hello lab 2
```

import_example.py

```
import random
random.randint(0,4)         # trebuie specificat numele modulului

import random as rand       # folosire alias pentru numele modulului
rand.randint(0,4)           # trebuie specificat alias-ul
```

```
from random import *        # import tot continutul modulului
randint(0,4)                # nu mai trebuie specificat numele modulului

from random import randint  # import doar randint
randint(0,4)
```

Ce este un thread?

Sistemele de calcul moderne sunt capabile de a executa mai multe operații în acelasi timp. Sistemul de operare este cel care permite rularea mai multor aplicații simultan, dar această idee se poate extinde și la nivelul unei aplicații. De exemplu, o aplicație ce rulează un stream video online trebuie simultan să *citească* conținutul video de pe rețea, să îl *decomprime*, să *actualizeze* display-ul local cu aceste informații etc. Spunem că aplicațiile ce oferă aceste capabilități constituie un software concurent.

Deci ce este concurența? **Concurența** este proprietatea unei logici de program de a putea executa **simultan** un set de task-uri. **Paralelismul** reprezintă o metodă de implementare a acestei paradigme de programare ce permite rularea unui set de task-uri într-un mod care utilizează core-uri multiple, procesoare multiple sau chiar mai multe mașini (într-o structură de tip cluster de exemplu).

Thread-urile reprezintă o metodă de implementare a concurenței, fiind fire de execuție create (*spawned*) în cadrul unui program principal (*process*) ce execută concurent task-uri definite de programator. Un fir de execuție este parte a unui proces, iar implementarea diferă de la un sistem de operare la altul. Mai multe thread-uri pot exista în cadrul aceluiași proces, ele partajând anumite resurse: memorie, descriptori I/O etc. În această privință thread-urile diferă de procese prin faptul că variabilele globale pot fi accesate de către toate thread-urile unui proces și pot servi ca mediu de comunicație între thread-uri. Fiecare thread are totuși și un set propriu de variabile locale. Din acest motiv thread-urile mai sunt numite și *lightweight processes*.

Clase și obiecte în Python

Trebuie subliniat că în Python, cuvântul "obiect" nu se referă neapărat la instanța unei clase. Clasele [https://docs.python.org/3.8/tutorial/classes.html] în sine sunt obiecte, iar, în sens mai larg, în Python toate tipurile de date sunt obiecte. Există tipuri de date care nu sunt clase: numerele întregi, listele, fișierele.

O clasă, în sensul C++/Java, se crează în Python prin folosirea cuvantului cheie `class`. Exemplul de mai jos creează un obiect de tip *class* cu numele *ClassName*. Acesta este echivalent cu o clasă C++/Java numită *ClassName*. *Interiorul clasei* poate conține definiții de metode sau clase și atribuiri de variabile. Clasa este derivată din *SuperClass1* și din *SuperClass2*. Spre deosebire de Java, numele fișierului sursă nu trebuie să fie la fel cu al vreunei clase definite în el.

```
class ClassName (SuperClass1, SuperClass2):
    [interiorul clasei]
```

Clasele suportă **multiple inheritence** și nu există un contract propriu-zis pentru interfețe. Pentru a crea clase abstracte există modulul abc (Abstract Base Classes) [http://docs.python.org/library/abc.html]. Pentru metodele pe care vreți să le considerați abstracte puteți transmite excepția `NotImplementedError` sau puteți adăuga în corpul funcției doar keyword-ul `pass`. Pentru a compara conceptele de programare orientată pe obiect în Python cu ceea ce sunteți familiari din Java vă recomandăm linkul OOP in Python vs Java [https://realpython.com/oop-in-python-vs-java/].

În lucrul cu clase, trebuie avute în vedere următoarele reguli:

- Primul argument pentru metodele unei clase este întotdeauna obiectul sursă, numit **`self`**, echivalent-ul lui `this`.
- Când ne referim la membrii clasei, trebuie să folosim `self.membru`, într-un mod asemanator cu folosirea "this" din Java (doar că în Python este obligatoriu să folosim `self` nu doar pentru a face distincție între câmpurile clasei și parametrii/variabilele cu aceleași nume din funcții).
- Metoda specială `__init__()` este apelată la instanțierea clasei și poate fi considerată un **constructor**. Definirea metodelor `__init__()` este opțională.

- Metoda specială `__del__()` este apelată când nu mai sunt referințe la acest obiect și poate fi asemuită cu un **destructor**. Definirea metodelor `__del__()` este opțională.
- În cazul moștenirii, în metoda `__init__()` trebuie întâi apelat `__init__()`-ul claselor părinte.
- Implicit toate câmpurile si metodele claselor sunt publice. Pentru a declara un câmp/metodă privată numele acesteia trebuie prefixat cu __ (mai multe detalii puteți afla aici [https://www.bogotobogo.com/python/python_private_attributes_methods.php]).
- Instanțierea se face prin apelarea obiectului clasă, posibil cu argumente.

class_example.py

```
class Student:
    """  O clasa care reprezinta un student. Comentariile docstring se pun dupa declaratie :) """

    def __init__(self, name, grade=5):   # constructor cu parametru default; echivalent cu mai multi constructori overloaded
        self.name = name                 # campurile clasei pot fi declarate oriunde!
        self.change_grade(grade)         # apelul unei metode a clasei

    def change_grade(self, grade):       # primul parametru este intotdeauna 'self'
        self.grade = grade               # adauga nou camp clasei

x = Student("Alice")
y = Student("Bob", 10)
x.change_grade(8)
```

Clasele Python pot avea membri statici. În cazul câmpurilor, ele sunt declarate în afara oricărei metode a clasei. Pentru metode avem două variante: una folosind decoratorul [https://realpython.com/primer-on-python-decorators/#decorating-classes] @staticmethod, cealaltă folosind funcția built-in staticmethod [https://docs.python.org/3.8/library/functions.html#staticmethod]. Observați că metodele statice nu au parametrul `self`.

static_example.py

```
class Util:
    x = 2                    # camp static

    @staticmethod            # metoda statica
    def do_stuff():
        print ("stuff")

    def do_otherstuff():     # alta varianta de a declara o metoda statica
        print ("other stuff")
    do_otherstuff = staticmethod(do_otherstuff)

print (Util.x)
Util.do_stuff()
Util.do_otherstuff()
```

Clase Python pe scurt:

- trebuie să folosiți `self.nume_funcție` sau `self.nume_variabila`
- o clasă poate moșteni mai multe clase
- `__init__()` este numele constructorului. Puteți avea un singur constructor, pentru a simula mai mulți contructori folosiți parametri default.
- puteți avea metode și variabile statice
- nu aveți *access modifiers*
- instanțiere: `nume_instanta = NumeClasa(argumente_constructor)`

Programare concurentă în Python

În Python, programarea concurentă este facilitată de modulul threading [http://docs.python.org/3/library/threading.html]. Acest modul oferă clasa Thread [http://docs.python.org/3/library/threading.html#thread-objects], care permite crearea și managementul thread-urilor, precum și o serie de clase (Condition [http://docs.python.org/3/library/threading.html#condition-objects], Event [http://docs.python.org/3/library/threading.html#event-objects], Lock

[http://docs.python.org/3/library/threading.html#lock-objects], RLock
[http://docs.python.org/3/library/threading.html#rlock-objects], Semaphore
[http://docs.python.org/3/library/threading.html#semaphore-objects], Barrier
[https://docs.python.org/3/library/threading.html#barrier-objects]) care oferă modalități de sincronizare și comunicare între thread-urile unui program Python.

Thread-uri

Un fir de execuție concurentă este reprezentat în Pyhton de clasa *Thread*. Cel mai simplu mod de a specifica instrucțiunile care se doresc a fi rulate concurent, este de a apela constructorul lui *Thread* cu numele unei funcții care conține aceste instrucțiuni, precum în exemplul următor. Pornirea thread-ului se face apoi cu metoda *start()*, iar pentru a aștepta terminarea execuției thread-ului se folosește metoda *join()*.

exemplul1.py

```
from threading import Thread

def my_concurrent_code(nr, msg):
    """ Functie care va fi rulata concurent """
    print ("Thread", nr, "says:", msg)

# creeaza obiectele corespunzatoare thread-urilor
t1 = Thread(target = my_concurrent_code, args = (1, "hello from thread"))
t2 = Thread(target = my_concurrent_code, args = (2, "hello from other thread"))

# porneste thread-urile
t1.start()
t2.start()

# executia thread-ului principal continua de asemenea
print ("Main thread says: hello from main")

# asteapta terminarea thread-urilor
t1.join()
t2.join()
```

Se folosește parametrul *target* al constructorului pentru a pasa numele funcției concurente și, opțional, pot fi folosiți parametrii *args* sau *kwargs* pentru a specifica argumentele funcției concurente, dacă ele există. *args* este folosit pentru a trimite argumentele funcției concurente ca un tuplu, iar *kwargs* este folosit pentru a trimite argumentele ca un dicționar. Pentru a diferenția un tuplu cu un singur element de folosirea obișnuită a parantezelor se utilizează următoarea sintaxă:

```
# t contine int-ul 42
t = (42)
# t contine un tuplu cu un singur element
t = (42,)
```

Crearea unui obiect Thread nu pornește execuția thread-ului. Acestu lucru se întâmplă doar după apelul metodei ***start()***.

O metodă alternativă de a specifica instrucțiunile care se doresc a fi rulate concurent este de a crea o subclasă a lui *Thread* care suprascrie metoda *run()*. Se poate de asemenea suprascrie și metoda *__init__()* (constructorul) pentru a primi argumentele cu care vor fi inițializate câmpurile proprii subclasei. Dacă optați pentru această abordare nu este indicat să suprascrieți alte metode ale clasei *Thread*, decât constructorul și *run()*.

exemplul2.py

```
from threading import Thread

class MyThread(Thread):
    """ Clasa care incapsuleaza codul nostru concurent """
    def __init__(self, nr, msg):
        Thread.__init__(self)
        self.nr = nr
        self.msg = msg

    def run(self):
        print ("Thread", self.nr, "says:", self.msg)

# creeaza obiectele corespunzatoare thread-urilor
```

```
t1 = MyThread(1, "hello from thread")
t2 = MyThread(2, "hello from other thread")

# porneste thread-urile
t1.start()
t2.start()

# executia thread-ului principal continua de asemenea
print ("Main thread says: hello from main")
```

La suprascrierea constructorului clasei *Thread* nu uitați să apelați și constructorul clasei de bază.

Interpretorul cel mai popular de Python (CPython) folosește un lock intern (GIL - Global Interpreter Lock) pentru a simplifica implementarea unor operații de nivel scăzut (managementul memoriei, apelul extensiilor scrise în C etc.). Acest lock permite execuția unui singur thread în interpretor la un moment dat și limitează paralelismul și performanța thread-urilor Python. Mai multe detalii despre GIL puteți găsi în această prezentare [http://www.dabeaz.com/python/UnderstandingGIL.pdf].

Pe lângă clasa *Thread* și clasele de sincronizare, modulul threading [https://docs.python.org/3/library/threading.html] mai conține și o serie de funcții ce ofera informații despre threadurile active. Python 3 a adăugat pe parcursul versiunilor sale noi astfel de metode utile. Observați că spre deosebire de Java, acestea sunt funcții definite în modul, nu apelate din cadrul vreunei clase.

```
>>> import threading
>>> threading.current_thread()
<_MainThread(MainThread, started 4566906304)>
>>> threading.active_count()
1
>>> threading.enumerate()
[<_MainThread(MainThread, started 4566906304)>]
>>> # Functii adaugate in Python 3
>>> threading.get_ident()
4566906304
>>> threading.get_native_id() # identificator folosit de kernel
3137981
>>> threading.main_thread()
<_MainThread(MainThread, started 4566906304)>
>>> # One-liner pentru crearea si pornirea unui thread
>>> threading.Thread(target=lambda a: print("ASC, lab %d", a), args=([2])).start()
ASC, lab %d 2
```

Elemente de sincronizare

Pentru ca un program concurent să funcționeze corect este nevoie ca firele sale de execuție să coopereze în momentul în care vor să acceseze date partajate. Această cooperare se face prin intermediul partajării unor elemente de sincronizare care pun la dispoziție un API ce oferă anumite garanții despre starea de execuție a thread-urilor care le folosesc.

Thread

Pe lângă facilitățile de creare a noi fire de execuție, obiectele de tip *Thread* reprezintă și cele mai simple elemente de sincronizare, prin intermediul metodelor *start()* și *join()*.

Metoda *start()* garantează că toate rezultatele thread-ului care o apelează (să-l numim *t1*), până în punctul apelului, sunt disponibile și în thread-ul care va porni (să-l numim *t2*). A se observa că nu se oferă nici un fel de garanție despre rezultatele lui *t1* care urmează după apel. *t2* nu poate face nici o presupunere în acest caz, fără a folosi alte elemente de sincronizare.

Metoda *join()* garantează thread-ului care o apelează (să-l numim *t1*) că thread-ul asupra căreia este apelată (să-l numim *t2*) s-a terminat și nu mai accesează date partajate. În plus toate rezultatele lui *t2* sunt disponibile și pot fi folosite de către *t1*. A se observa că, față de metoda *start()*, metoda *join()* blochează execuția thread-ului care o apelează (*t1*) până când *t2* își termină execuția. Spunem că *join()* este o metodă blocantă.

Lock

Lock-ul este un element de sincronizare care oferă acces exclusiv la porțiunile de cod protejate de către lock (cu alte cuvinte definește o secțiune critică). Python pune la dispoziție clasa *Lock* pentru a lucra cu acest element de

sincronizare. Un obiect de tip *Lock* se poate afla într-una din următoarele două stări: **blocat** sau **neblocat**, implicit, un obiect de tip *Lock* fiind creat în starea **neblocat**. Sunt oferite două operații care controlează starea unui lock: *acquire()* și *release()*.

- *acquire()* va trece lock-ul în starea blocat. Dacă lock-ul se afla deja în starea blocat, thread-ul care a apelat *acquire()* se va bloca până când lock-ul este eliberat (pentru a putea fi blocat din nou).
- *release()* este cea care trece lock-ul în starea deblocat. Cele două metode garantează că un singur thread poate deține lock-ul la un moment dat, oferind astfel posibilitatea ca un singur thread să execute secțiunea de cod critică. O altă garanție a lock-ului este că toate rezultatele thread-ului care a efectuat *release()* sunt disponibile și pot fi folosite de următoarele thread-uri care execută *acquire()*.

Lock-ul este utilizat în majoritatea cazurilor pentru a **proteja accesul la structuri de date partajate**, care altfel ar putea fi modificate de un fir de execuție în timp ce alte fire de execuție încearcă simultan să citească sau să modifice și ele aceeași structură de date. Pentru a rezolva această situație, porțiunile de cod care accesează structura de date partajată sunt încadrate între apeluri *acquire()* și *release()* pe **același** obiect *Lock* partajat de toate thread-urile care vor sa acceseze structura.

Spre deosebire de un mutex (ex: pthread_mutex [https://computing.llnl.gov/tutorials/pthreads/#Mutexes]), în Python, metodele *acquire()* și *release()* pot fi apelate de thread-uri diferite. Cu alte cuvinte un thread poate face *acquire()* și alt thread poate face *release()*. Datorită acestei diferențe subtile nu este recomandat să folosiți un obiect *Lock* în acest mod. Pentru a reduce confuziile și a obține același efect se poate folosi un obiect *BoundedSemaphore* inițializat cu valoarea 1.

Exemplul de mai jos prezintă folosirea unui lock pentru a proteja accesul la o listă partajată de mai multe thread-uri.

exemplul3.py

```
from threading import Lock, Thread

def inc(lista, lock, index, n):
    """ Incrementeaza elementul index din lista de n ori """
    for i in range(n):
        lock.acquire()
        lista[index] += 1
        lock.release()

def dec(lista, lock, index, n):
    """ Decrementeaza elementul index din lista de n ori """
    for i in range(n):
        lock.acquire()
        lista[index] -= 1
        lock.release()
# lista si lock-ul care o protejeaza
my_list = [0]
my_lock = Lock()

# thread-urile care modifica elemente din lista
t1 = Thread(target = inc, args = (my_list, my_lock, 0, 100))
t2 = Thread(target = dec, args = (my_list, my_lock, 0, 100))

# lista inainte de modificari
print (my_list)

t1.start()
t2.start()

t1.join()
t2.join()

# lista dupa modificari
print (my_list)
```

Puteți folosi construcția *with* pentru a delimita o secțiune critică astfel:

```
def inc(lista, lock, index, n):
    for i in range(n):
        with lock:
            lista[index] += 1
```

Obiecte thread-safe

Anumite operații pe obiecte din Python (tipuri primitive, liste, dictionare) sunt atomice, asta însemnând că nu trebuie protejate prin obiecte de sincronizare dacă acel obiect este partajat de mai multe threaduri. Lista lor o găsiți aici [https://docs.python.org/3/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe].

Listele de exemplu sunt protejate de către lock-ul global al interpretorului (GIL) și sunt considerate thread-safe. Atenție, asta nu înseamnă că orice operație care are legătură cu o listă este safe! În exemplul din secțiunea Lock operația de modificare a elementului listei nu este atomică și a fost nevoie să protejăm prin lock-uri pentru a nu corupe lista.

```
>>> my_list = [1,2]
>>> my_list.append(0) # safe
>>> my_list[1] += 1   # not safe, += nu este operatie atomica!
>>> my_list
[1, 3, 0]
>>> my_list.sort()    # safe
>>> my_list
[0, 1, 3]
```

RLock

RLock-ul (Re-entrant Lock) este un element de sincronizare similar cu Lock, dar care oferă posibilitatea de a accesa o secțiune critică de mai multe ori din același thread. Accesările structurilor de date partajate sunt încadrate între apelurile *acquire()* și *release()*, la fel ca și în cazul Lock-ului.

Intern sunt folosite concepte precum *owning thread* și *recursion level* pe lângă starea lock-ului de blocat/neblocat. În momentul în care un anumit thread blochează lock-ul, el, și numai el, îl poate re-bloca, fără să intre în deadlock și fără a-l elibera în prealabil. Pentru a elibera secțiunea/lock-ul, thread-ul va trebui să apeleze *release()* de un număr de ori egal cu numărul de apeluri *acquire()*.

RLock-ul devine util însă în momentul folosirii unor secțiuni critice imbricate, cauzate de funcții recursive, sau pur și simplu de organizarea codului în funcții multiple, care se apelează reciproc. În cazul folosiri RLock-ului nu mai este nevoie de urmărirea manuală stării lock-ului pentru a evita deadlock-ul ce apare la imbricarea secțiunilor critice definite de un Lock simplu.

exemplul4.py

```
from threading import RLock, Thread

def inc(lista, rlock, index, n):
    """ Incrementeaza elementul index din lista de n ori """
    rlock.acquire()
    if n > 0:
        lista[index] += 1               # incrementeaza o data
        inc(lista, rlock, index, n - 1) # incrementeaza recursiv de n-1 ori
    rlock.release()

def dec(lista, rlock, index, n):
    """ Decrementeaza elementul index din lista de n ori """
    rlock.acquire()
    if n > 0:
        lista[index] -= 1               # decrementeaza o data
        dec(lista, rlock, index, n - 1) # decrementeaza recursiv de n-1 ori
    rlock.release()

# lista si lock-ul care o protejeaza
my_list = [0]
my_lock = RLock()

# thread-urile care modifica elemente din lista
t1 = Thread(target = inc, args = (my_list, my_lock, 0, 100))
t2 = Thread(target = dec, args = (my_list, my_lock, 0, 100))

# lista inainte de modificari
print (my_list)

t1.start()
t2.start()

t1.join()
```

```
t2.join()

# lista dupa modificari
print (my_list)
```

Un exemplu despre diferența de funcționare între Lock și RLock la nivel de apelare a funcțiilor

Lock	***RLock***

`import threading`
 	
`lock = threading.Lock()`
 	
`print ('First try :', lock.acquire())`
`print ('Second try:', lock.acquire())`
 	
`print ("print this if not blocked...")`
	`import threading`
	
	`lock = threading.RLock()`
	
	`print ('First try :', lock.acquire())`
	`print ('Second try:', lock.acquire())`
	
	`print ("print this if not blocked...")`

Output	
`First try : True` `Second try:`	`First try : True` `Second try: print this if not blocked...`

Semaphore

Semaforul este un element de sincronizare cu o interfață asemănătoare Lock-ului (metodele *acquire()* și *release()*) însă cu o comportare diferită. Python oferă suport pentru semafoare prin intermediul clasei *Semaphore*.

Un *Semaphore* menține un contor intern care este decrementat de un apel *acquire()* și incrementat de un apel *release()*. Metoda *acquire()* nu va permite decrementarea contorului sub valoarea 0, ea blocând execuția thread-ului în acest caz până când contorul este incrementat de un *release()*. Metodele *acquire()* și *release()* pot fi apelate fără probleme de thread-uri diferite, această utilizare fiind des întâlnită în cazul semafoarelor.

Un exemplu clasic de folosire a semaforului este acela de a limita numărul de thread-uri care accesează concurent o resursă precum în exemplul următor:

exemplul5.py

```
from random import randint, seed
from threading import Semaphore, Thread
from time import sleep

def access(nr, sem):
    sem.acquire()
    print ("Thread-ul", nr, " acceseaza")
    sleep(randint(1, 4))
    print ("Thread-ul", nr, " a terminat")
    sem.release()

# initializam semaforul cu 3 pentru a avea maxim 3 thread-uri active la un moment dat
semafor = Semaphore(value = 3)

# stocam obiectele Thread pentru a putea face join
thread_list = []

seed() # seed-ul este current system time pentru generatorul de nr random

# pornim thread-urile
for i in range(10):
    thread = Thread(target = access, args = (i, semafor))
    thread.start()
    thread_list.append(thread)

# asteptam terminarea thread-urilor
for i in range(len(thread_list)):
    thread_list[i].join()
```

Exerciții

1. Descărcați și rulați cele 5 exemple oferite în laborator. Ce observați la fiecare dintre ele?
2. *Hello Thread* - creați și rulați threaduri urmărind cerințele din fișierul `task2.py` din scheletul de laborator.
3. *Coffee Factory* - problema producător-consumator folosind semafoare.
    - În schelet este dată o implementare sumară a clasei Coffee și a unei clase ExampleCoffee. Folosind ca model, clasa ExampleCoffee, realizați alte 3 implementări pentru următoarele tipuri de cafea: Espresso, Americano și Cappuccino, care vor trebui să moștenească clasa de bază Coffee.
    - CoffeeFactory funcționează ca un producător, iar User ca un consumator și vor trebui să realizeze operațiunile de produce și consume, mereu, fără blocaje.
    - Implementați toate aceste clase, precum și clasa Distributor, care deține un buffer limitat la un număr fix de cafele, precum și de elementele de sincronizare necesare.
    - **Important:** Implementarea trebuie să funcționeze pentru **oricâți** producători și **oricâți** consumatori, numere care vor fi propuse de către fiecare asistent, în momentul verificării.
    - **Hint:** De câte semafoare am avea nevoie?
4. Implementați problema filozofilor.
    - Se consideră mai mulți filozofi ce stau în jurul unei mese rotunde. În mijlocul mesei este o farfurie cu spaghete. Pentru a putea mânca, un filozof are nevoie de două bețisoare. Pe masă există câte un bețișor între fiecare doi filozofi vecini. Regula este că fiecare filozof poate folosi doar bețișoarele din imediata sa apropriere. Trebuie evitată situația în care nici un filozof nu poate acapara ambele bețișoare. Comportamentul tuturor filozofilor trebuie să fie identic.

Puteți găsi materiale ajutătoare în cadrul laboratorului 5 de la APD [https://ocw.cs.pub.ro/courses/apd/laboratoare/05]

Codul protejat (încadrat) de lockuri nu e bine să conțină *print-uri* sau instrucțiuni ce nu lucrează cu variabila partajată (e.g. *sleep*).

Nu este recomandat ca într-o aplicație concurentă să aveți print-uri, și nici lock-uri pe print-uri, acest lucru afectând comportarea threadurilor. În laborator se folosesc print-uri în scop de debugging, și vă recomandăm să folosiți *format* sau concatenare (+) pentru a obține afișări atomice.

Resurse

- PDF laborator
- Schelet laborator
- Soluție laborator

Referințe

- OOP in Python vs Java [https://realpython.com/oop-in-python-vs-java/]

Documentație module

- modulul threading [http://docs.python.org/3/library/threading.html] - Thread, Lock, Semaphore
- modulul _thread [https://docs.python.org/3/library/_thread.html#module-_thread]

Detalii legate de implementare

- What kinds of global value mutation are thread-safe [https://docs.python.org/3/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe]
- Implementarea obiectelor de sincronizare în CPython [http://dabeaz.blogspot.ro/2009/09/python-thread-synchronization.html]
- What is the Python Global Interpreter Lock (GIL)? [https://realpython.com/python-gil/]
- Grok the GIL: How to write fast and thread-safe Python [https://opensource.com/article/17/4/grok-gil]

- Understanding the Python GIL [http://www.dabeaz.com/python/UnderstandingGIL.pdf] (prezentare foarte bună și amuzantă)

Despre concurență și obiecte de sincronizare

- Versatilitatea Semafoarelor [http://preshing.com/20150316/semaphores-are-surprisingly-versatile/]
- Mecanisme de sincronizare threaduri în Python [http://effbot.org/zone/thread-synchronization.htm]
- Understanding Threading in Python [http://linuxgazette.net/107/pai.html]
- Little book of semaphores

asc/laboratoare/02.txt · Last modified: 2021/02/16 20:39 (external edit)