Катедра: „Информатика и софтуерни науки”
Дисциплина: „Приложен Изкуствен Интелект”

К У Р С О В А Р А Б О Т А

Тема:

Multithreading (Python)

Thread, daemon thread, join(), ThreadPoolExecutor, race conditions, synchronization, deadlock, producer-consumer

Десислава Емилова Милушева
ИСН, курс III, гр. 77,
Фак. № 471219007

Николай Валентинов Каймакански
ИСН, курс III, гр. 76,
Фак. № 471219072

Разработили: …….

/ Десислава Милушева /
/ Николай Каймакански /

Проверил: ……

/ Александър Ефремов /
/ Ивета Григорова /

София, 2022 г.

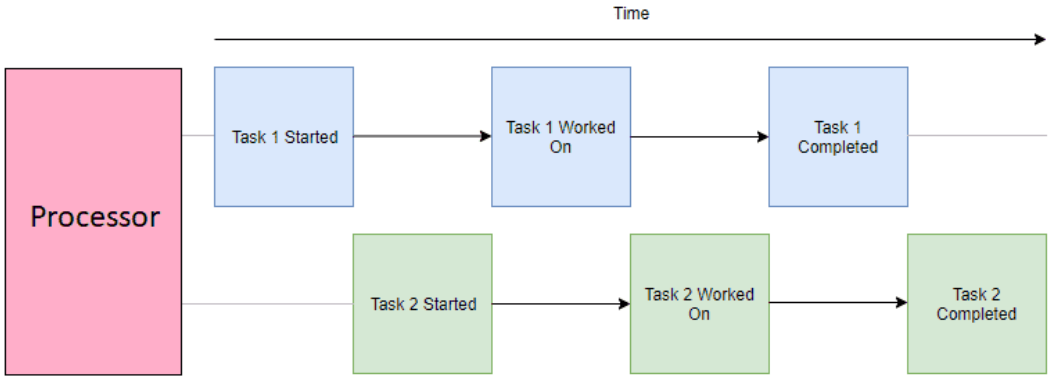
Съдържание

Част 1 - Въведение и приложение

1. Въведение

Изпълнението на множество задачи едновременно (т.н. multithreading) е способността на всяка операционна система да има повече от една програма, работеща в един и същи момент от времето. Например, можем да принтираме, докато редактираме или изтегляме имейла си, или да слушаме музика от музикалният плеър, докато разглеждаме снимки от галерията. В днешно време всеки компютър има повече от един процесор, но броят на едновременно изпълняваните процеси не е ограничен от броя на процесорите. Операционната система заделя отрязъци от време за всеки процес, създавайки впечатление за паралелна обработка. Многонишковите програми разширяват идеята за изпълнение на много процеси едновременно, като я спускат с едно ниво по-ниско: отделните програми ще изглежда да изпълняват множество задачи едновременно.

Python поддържа различни подходи свързани със "състезателното" програмиране (т.н. concurrent programming), включително програмиране с нишки, стартиране на подпроцеси и други различни трикове. В тази курсова работа целим да представим и разгледаме концепции, свързани с различни аспекти на едновременното програмиране, включително техники за програмиране с обща нишка и подходи свързани с това как един процесор например, може да постигне напредък по множество задачи, привидно едновременно в един и същи период от време.



2. Приложение

Обхватът на приложение на multithreading-а в Python е изключително голям и почти всяко приложение или програма използва повече от една нишка, по време на своето изпълнение, като така се оптимизира и самото действие, бързина и работа. Например, нека разгледаме случай на приложение, при който имаме компютър с едноядрен процесор. Това означава, че задачите, които трябва да бъдат изпълнени като част от приложение, не могат да постигнат напредък по едно и също време, тъй като процесорът е в състояние да работи само върху една задача в даден момент. Изпълнението на множество задачи едновременно означава, че процесорът извършва превключване на контекста, така че няколко задачи да могат да се изпълняват едновременно.

Част 2 - Термини и имплементация на Python

- Thread:

Нишката (thread) е отделен поток на изпълнение на даден процес. Това означава, че изпълнението на една програма ще може да бъде разделено на няколко части, които части се изпълняват едновременно. Може да мислим за нишката като с два (или повече) различни процесора, работещи в дадена програма, всеки от които изпълнява независима задача по едно и също време.

Нишките могат да се разглеждат като по-малки и леки процеси. Освен това те са най-малката конструкция, която може да се управлява от операционната система. Нишките са свързани с процеса, който ги е създал. Те нямат собствена памет, както има процеса. Вместо това, те споделят паметта на процеса, който ги е създал. Процесът винаги ще има поне една нишка, свързана с него, която се нарича основна нишка. Всички останали нишки, които бъдат създадени от процесът се наричат работни или фонови нишки. Тези нишки могат да извършват друга работа едновременно по дължината на основната нишка. Нишките, подобно на процесите, могат да работят една с друга на многоядрен процесор, а операционната система също може да превключва между тях чрез разделяне на времето. Когато стартираме нормално Python приложение, ние създаваме процес, както и основна нишка, която ще отговаря за изпълнението на нашето Python приложение.

```
# The simplest way to use a Thread is to instantiate it with
# a target function and call start() to let it begin working.

import threading
import time

from threading import Thread


def work(seconds):
    time.sleep(seconds)
    # thread worker function
    print('...' + threading.current_thread().getName() + ' is working...')


if __name__ == "__main__":
    counter_seconds = 5
    print('Start the threads after ' + str(counter_seconds) + ' seconds ...')
    thread_1 = threading.Thread(target=work, args=(counter_seconds,),
name="Thread-1")
    thread_2 = Thread(target=work, args=(counter_seconds,) , name="Thread-2")

    thread_1.start()
    thread_2.start()

    for x in range(1, 6):
        # sleep the main thread while counting from 1 to 5
        time.sleep(0.9)
        print('Main thread is counting: ' + str(x))
```

```
Start the threads after 5 seconds ...
Main thread is counting: 1
Main thread is counting: 2
Main thread is counting: 3
Main thread is counting: 4
Main thread is counting: 5
...Thread-1 is working...
...Thread-2 is working...
```

- Daemon thread:

В компютърните науки демонът е процес, който работи във фонов режим. Нишките на Python имат по-специфично значение за daemon. Нишката на демон ще се изключи незабавно, когато програмата излезе. Можем да считаме нишката на демона за нишка, която работи във фонов режим, без да се притесняваме, че ще я изключим. Ако програма изпълнява Threads, които не са демони, тогава програмата ще изчака тези нишки да завършат, преди да приключи. Нишките, които са демони, обаче просто се убиват, където и да се намират, когато програмата излиза.

За пример нека си представаим приложение за обработка на текст (Text Editor App). В него имаме нишка, която запазва нашата работа във файл на всеки няколко минути. Ако изведнъж решим да затворим приложението, ние няма да се интересуваме от това дали тази нишка, която запазва нашата работа и работи на заден план, все още се изпълнява. Не искаме да чакаме тази нишка да приключи, за да затворим приложението, но искаме нашата работа да бъде запазена. Така затваряйки приложението, въпросната нишка, която запазва информацията ще продължи да работи още малко като демон нишка на заден план, но това по никакъв начин няма да е видимо за нас като потребители.

Демон нишката е фонова нишка, която не пречи на приложението да излезе, ако основната нишка приключи. Нека разгледаме следния пример:

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG, format='(%(threadName)-9s) %(message)s',)

def start_thread():
   logging.debug('Starting')
   logging.debug('Exiting')

def start_daemon():
   logging.debug('Starting')
   time.sleep(5)
   logging.debug('Exiting')

if __name__ == '__main__':

  thread = threading.Thread(name='non-daemon', target=start_thread)

  daemon = threading.Thread(name='daemon', target=start_daemon)
  daemon.setDaemon(True)

  daemon.start()
  thread.start()
```

```
(daemon   ) Starting
(non-daemon) Starting
(non-daemon) Exiting
```

Обикновено нашата основна програма изчаква, докато всички други нишки завършат работата си. Въпреки това, понякога програмите създават демон нишка, която работи без да блокира излизането на основната програма. Използването на демонни нишки е полезно, когато няма лесен начин за прекъсване на основната нишката или, когато оставим основната нишката да умре в средата на работата си, а не искаме да губи или повреждa данните, с който работи програмата, както в посочения пример с Text Editor App. За да

обозначим нишка като демон нишка, ние извикваме нейния метод setDaemon() с булев аргумент True. Настройката по подразбиране за всяка нишка не е демон, за това трябва изрично да го зададем И така, задавайки True, се включва режима на демон на нишката.

Както можем да видим от изхода на програмата, няма съобщение "(daemon) Exiting" от демон нишката, тъй като всички нишки, които не са демони (включително основната нишка) излизат преди демона нишката да се събуди от петсекундния си режим на заспиване. Съответно тя продължава да работи на заден фон след тези пет секунди, но тък като основната нишка е приключила, не може да видим съобщението "(daemon) Exiting". Ако обаче махнем time.sleep(5) във функцията на демон нишката start_daemon(), демонът също ще излезе и резултатът ще изглежда така:

```
(daemon    ) Starting
(daemon    ) Exiting
(non-daemon) Starting
(non-daemon) Exiting
```

- join():

Функцията join() се използва, когато искаме да кажем на една нишка да изчака друга нишка да завърши. За да покажем как работи тази концепция, нека да използваме предния пример и да използваме join() метода, за да изчакаме, докато демон нишка завърши работата си.

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG, format='(%(threadName)-9s) %(message)s',)

def start_thread():
   logging.debug('Starting')
   logging.debug('Exiting')

def start_daemon():
   logging.debug('Starting')
   time.sleep(5)
   logging.debug('Exiting')

if __name__ == '__main__':

  thread = threading.Thread(name='non-daemon', target=start_thread)

  daemon = threading.Thread(name='daemon', target=start_daemon)
```

```
daemon.setDaemon(True)

daemon.start()
thread.start()

daemon.join()
thread.join()
```

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
(daemon    ) Exiting
```

При изпълнението на програмата ще види, че изходът на демон нишката е около 5 секунди след изхода на не-демон нишката. По подразбиране join() блокира за неопределено време. В нашата случай, join() блокира извикващата нишка (основната нишка), докато нишките daemon и thread, чийто метод join() е извикан, не бъдат прекратени.

- ThreadPoolExecutor:

Подобно на изпълнителите на пул на процеси, библиотеката concurrent.futures предоставя реализация на абстрактния клас Executor за работа с нишки, наречени ThreadPoolExecutor. Вместо да поддържа пул от работни процеси, както прави пул от процеси, изпълнителят на пул от нишки ще създаде и поддържа пул от нишки, на който след това можем да изпратим работа. Докато пулът от процеси по подразбиране създава един работен процес за всяко ядро на процесора, с което разполага нашата машина, определянето на колко работни нишки да се създаде е малко по-сложно. Вътрешно формулата за броя на нишките по подразбиране е min(32, os.cpu_count() + 4). Това кара максималната (горна) граница на работните нишки да е 32, а минималната (долната) да е 5. Горната граница е зададена на 32, за да се избегне създаването на изненадващ брой нишки на машини с големи количества процесорни ядра (запомнете , нишките са скъпи за създаване и поддръжка). Долната граница е зададена на 5, тъй като на по-малки машини с 1–2 ядра, въртенето само на няколко нишки няма вероятност да подобри много производителността. Често има смисъл да създадете няколко повече нишки от наличните ви процесори за работа, свързана с I/O. Например, на 8-ядрена машина горната формула означава, че ще създадем 12 нишки. Докато само 8 нишки могат да се изпълняват едновременно, можем да поставим други нишки на пауза в изчакване на I/O да приключи, оставяйки нашата работа да ги възобнови, когато I/O приключи.

** submit ** методът планира подадената му функция за изпълнение и връща обект от тип future. Този обект капсулира изпълнението на функцията.

** Примерно приложение за сваляне на изображения, ползвайки ThreadPoolExecutor

```
import time
import requests
from concurrent.futures import ThreadPoolExecutor

image_urls = [
    'https://images.unsplash.com/photo-1516117172878-fd2c41f4a759',
    'https://images.unsplash.com/photo-1532009324734-20a7a5813719',
    'https://images.unsplash.com/photo-1524429656589-6633a470097c',
    'https://images.unsplash.com/photo-1530224264768-7ff8c1789d79',
    'https://images.unsplash.com/photo-1564135624576-c5c88640f235',
    'https://images.unsplash.com/photo-1541698444083-023c97d3f4b6',
    'https://images.unsplash.com/photo-1522364723953-452d3431c267',
    'https://images.unsplash.com/photo-1513938709626-033611b8cc03',
    'https://images.unsplash.com/photo-1507143550189-fed454f93097',
    'https://images.unsplash.com/photo-1493976040374-85c8e12f0c0e'
]

start = time.perf_counter()

# ** Downloading images synchronous one by one **
for img_url in image_urls:
    img_bytes = requests.get(img_url).content
    img_name = img_url.split('/')[3]
    img_name = f'{img_name}.jpg'
    time.sleep(1)
    with open(img_name, 'wb') as img_file:
        img_file.write(img_bytes)
        print(f'{img_name} was downloaded...')


# ** Downloading images asynchronously with multithreading
# def download_img(img_url):
#     img_bytes = requests.get(img_url).content
#     img_name = img_url.split('/')[3]
#     img_name = f'{img_name}.jpg'
#     time.sleep(1)
#     with open(img_name, 'wb') as img_file:
#         img_file.write(img_bytes)
#         print(f'{img_name} was downloaded...')
#
# with ThreadPoolExecutor() as pool:
#     pool.map(download_img, image_urls)

end = time.perf_counter()
print(f'Finished in {end - start} seconds')
```

```
photo-1516117172878-fd2c41f4a759.jpg was downloaded...
photo-1532009324734-20a7a5813719.jpg was downloaded...
photo-1524429656589-6633a470097c.jpg was downloaded...
photo-1530224264768-7ff8c1789d79.jpg was downloaded...
photo-1564135624576-c5c88640f235.jpg was downloaded...
photo-1541698444083-023c97d3f4b6.jpg was downloaded...
photo-1522364723953-452d3431c267.jpg was downloaded...
photo-1513938709626-033611b8cc03.jpg was downloaded...
photo-1507143550189-fed454f93097.jpg was downloaded...
photo-1493976040374-85c8e12f0c0e.jpg was downloaded...
Finished in 27.660984129004646 seconds
```

- Race conditions:

В повечето практични многонишкови приложения две или повече нишки трябва да споделят достъп до едни и същи данни или ресурс под формата например на файл. Но, ако две нишки имат достъп до един и същ обект и всяка извиква метод, който променя състоянието на обекта, тогава нишките могат да си пречат и в зависимост от реда, в който са били достъпни данните, могат да се получат повредени обеми от данни. Такава ситуация често се нарича състезателно състояние (т.н. race condition).

```
import threading
import time

x = 10

def increment(increment_by):
    global x

    local_counter = x
    local_counter += increment_by
    time.sleep(1)

    x = local_counter
    print(f'{threading.current_thread().name} increments x by {increment_by}, x:
{x}')

# creating threads
t1 = threading.Thread(target=increment, args=(5,))
t2 = threading.Thread(target=increment, args=(10,))

# starting the threads
t1.start()
t2.start()

# waiting for the threads to complete
t1.join()
t2.join()
```

```
print(f'The final value of x is {x}')
```

```
Thread-1 increments x by 5, x: 15
Thread-2 increments x by 10, x: 20
The final value of x is 20
```

На първите два реда сме импортнали модулит threading за нишките и time за времето. След това сме инициализирали променливата x=10, която ще играе ролята на споделен ресурс в нашия пример за нишки t1 и t2. Двете нишки се създават като се дефинира целевата функция increment. Нишки t1 и t2 ще се опитат да променят/ увеличат стойността на x във функцията за нарастване increment с 5 и с 10. Когато дадем start, нишките ще се инициирате, докато join ще изчака да завършат изпълнението си, тъй като те ще заспиват за по 1 секунда във функцията за увеличаване increment.

Както виждаме крайният резултат е 20, а не 25, както се очаква след като съберем: 10 + 5 + 10 = 25. Това се получава пореди наличието на състезателно състояние между нишките, които достъпват общата променлива x.

Едно бързо решение на този проблем може да е, когато добавим механизъм за синхронизация (между две или повече нишки) и заключим споделения ресурс.Тоест, един процес може да заключи споделения ресурс и да го направи недостъпен за други, ако работи с него в дадения момент. Заключването има две състояния:

- lock – означава, че критичната секция е заета, в двоичен вид, т.е. 1
- unlock – означава, че критичната секция е свободна, в двоичен вид, т.е. 0.

```
import threading
from threading import Lock
import time

x = 10

def increment(increment_by,lock):
   global x

   lock.acquire()

   local_counter = x
   local_counter += increment_by

   time.sleep(1)

   x = local_counter
   print(f'{threading.current_thread().name} increments x by {increment_by}, x:
{x}')

   lock.release()
```

```
lock = Lock()

# creating threads
t1 = threading.Thread(target=increment, args=(5,lock))
t2 = threading.Thread(target=increment, args=(10,lock))

# starting the threads
t1.start()
t2.start()

# waiting for the threads to complete
t1.join()
t2.join()

print(f'The final value of x is {x}')
```

```
Thread-1 increments x by 5, x: 15
Thread-2 increments x by 10, x: 25
The final value of x is 25
```

За да избегнем състеателното състоянието, импортирахме класа Lock на модула за threading и създадохме негов обект, наречен lock. Lock има методи, а именно acquire() и release(), които заключват и отключват дадения ресурс. Например, функцията за нарастване на t1 заключва ресурса x, докато го увеличава с 5 и така t2 не може да променя или да се намесва в операцията, докато ресурсът не бъде отключен от t1. Първо t1 завършва своята операция и после и t2 увеличава x, следователно получаваме желаната стойност 25 като резултат.

- Synchronization:

За да се избегне състезателното състояние обяснено в предната подточка и, за да няма повредени данни, които са били споделени между множество нишки, трябва да се използва подход за синхронизиран достъпа или по-точно синхронизация (т.н. synchronization). Пример за такъв подход е ползването на mutex (mutual exclusion), както е показано в примера:

```
import time
import random
import threading
from concurrent.futures import ThreadPoolExecutor

# Synchronization example using mutex and ThreadPoolExecutor:

lock = threading.Lock()
```

```python
def using_the_wc(lockWC, somebody):
    lockWC.acquire()
    print(f'WC is occupied by {somebody}!')
    start = time.perf_counter()
    time.sleep(random.randint(1, 5))
    end = time.perf_counter()
    print(f'{somebody} finished in {end - start} seconds')
    lockWC.release()

with ThreadPoolExecutor() as pool:
    people = ['Billy', 'Johny', 'Tony', 'Bobby']
    for someone in people:
        pool.submit(using_the_wc, lock, someone)
```

```
WC is occupied by Billy!
Billy finished in 4.004215107008349 seconds
WC is occupied by Johny!
Johny finished in 5.004785075012478 seconds
WC is occupied by Tony!
Tony finished in 2.0021247399999993 seconds
WC is occupied by Bobby!
Bobby finished in 2.001107422009227 seconds
```

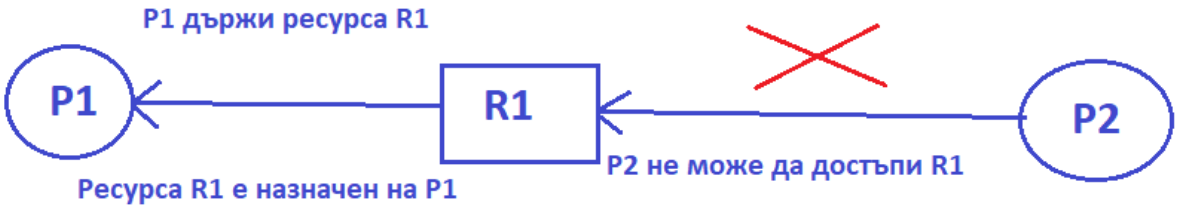
- Deadlock:

Мъртва хватка (т.н. deadlock), един от най-често срещаните проблеми с паралелността. В областта на компютърните науки, мъртва хватка се отнася до специфична ситуация в multithreading програмирането, при която не може да се постигне напредък и програмата се заключва в текущото си състояние. В повечето случаи това явление е причинено от липса или неправилна координация между различни обекти за заключване (за целите на синхронизиране на нишки).

** Предпоставки за Deadlock:

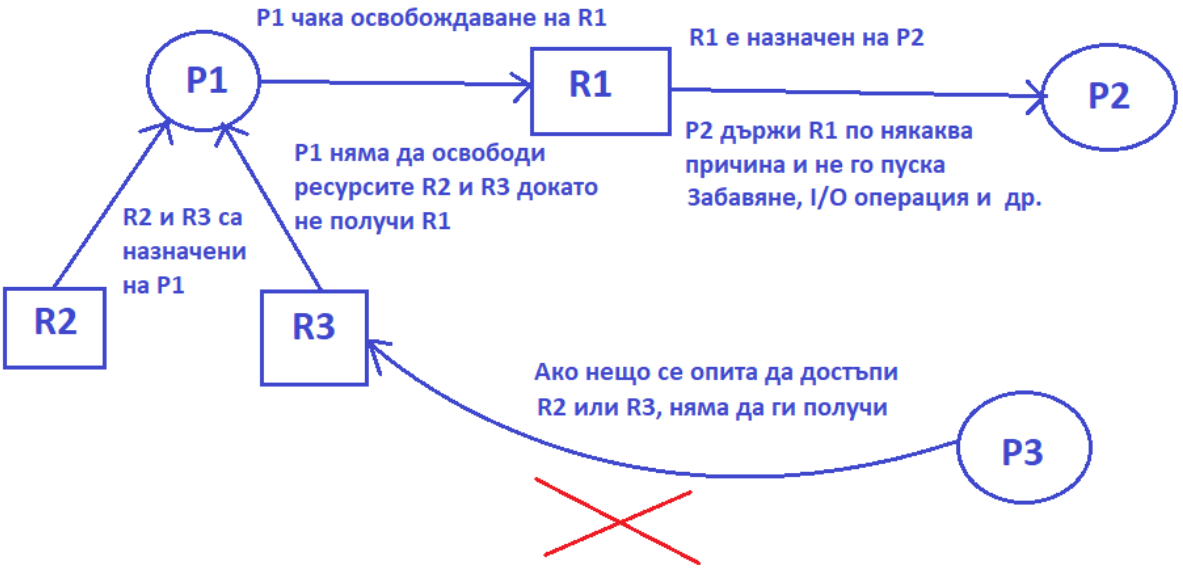
1. Mutual exclusion

Един или повече ресурси са несподеляеми. Само 1 процес може да ги достъпва в даден момент.



2. Hold and wait

Процес държи поне един ресурс и чака други ресурс/и, да се освободят.

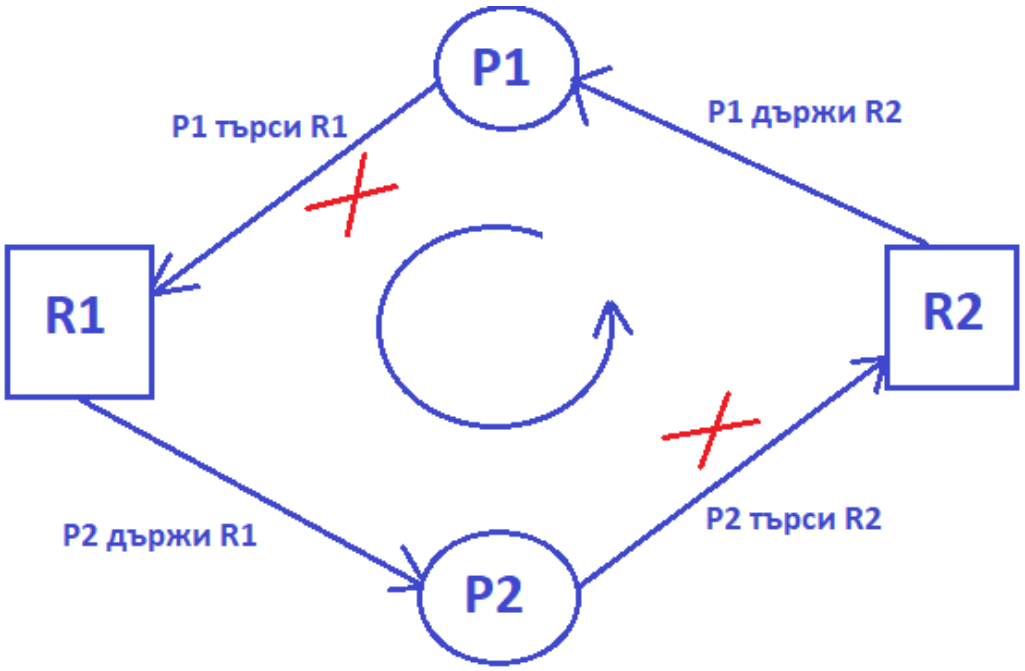


3. No Preemption

Операционната система не може да прекъсне работата на процеса и да му изземе ресурса, който използва

4. Circular waiting

Няколко процеса са си заключили ресурсите в циркулярна форма и чакат другия да ги освободи.



- Когато всеки процес бъде блокиран, не може да продължи и четирите предпоставки са изпълнени едновременно се получава Deadlock. За да го предотвратим трябва да избегнем поне една от предпоставките му.

** Deadlock vs Starvation:

- при starvation ниско-приоритетните процеси биват блокирани от тези с по-висок приоритетните. Ресурсите биват използвани постоянно от процесите с по-висок приоритет. Когато на процес с по-нисък приоритет постоянно му се отказват жизнено важни ресурси, той "изгладнява". Това може да бъде причинено от грешки в schedualing-а или в mutual exclusion алгоритъм-а, но също така и нарочно чрез т.н. "Denial of service attack". За да се избегне starvation трябва индивидуално да се менежират ресурсите
и да не се налага стриктна приоритизация.

** Livelock:

- когато 2 нишки са толкова заети да отговарят една на друга,
че не остава време да продължат с фундаменталната задача на програмата
може да се получи Livelock (което пък от своя страна може да бъде причина за starvation). Те не са изцяло блокирани, а минават една след друга в активно състояние, без да могат да завършат. Например при отваряне на апликация, потребителят получава "Not responding" и след известно време апликацията бавно започва да работи.. (евентуално).

** The Dining Philosophers Problem

5 философа седят на кръгла маса. Пред всеки има по 1 купа спагети и между всеки двама има по един чопстик. Философът има две състояния: да мисли и да яде. Когато мисли не комуникира с останалите философи, а когато огладнее се опитва да яде от купата пред него. За да може да яде, обаче са му необходими два чопстика - един от ляво и един от дясно. Не може да вземе двата чопстика едновременно, само един по един. Не може да вземе чопстик, който вече е взет от съседа. Когато философът приключи с яденето оставя чопстиковете и започва отново да мисли.

```
from concurrent.futures import ThreadPoolExecutor
from threading import Lock
import time
import random


class DiningPhilosophers:
    def __init__(self, number_of_phils, meals_to_eat):
        self.meals = [meals_to_eat for _ in range(number_of_phils)]
        self.chopsticks = [Lock() for _ in range(number_of_phils)]

    def serveMeals(self, current_philosopher):
        first_chopstick = current_philosopher
        second_chopstick = (current_philosopher + 1) % 5
# яж докато има на масата
```

```
        while self.meals[current_philosopher] > 0:
            # thinking state
            time.sleep(random.random()*5)
            self.chopsticks[first_chopstick].acquire()
            time.sleep(random.random())

            if not self.chopsticks[second_chopstick].locked():
                self.chopsticks[second_chopstick].acquire()
                print(f'Philosopher {current_philosopher} starts eating...')
                # eating state
                time.sleep(random.random()*5)
                self.meals[current_philosopher] -= 1
                self.chopsticks[first_chopstick].release()
                self.chopsticks[second_chopstick].release()
                print(f'Philosopher {current_philosopher} stops eating.
{self.meals[current_philosopher]} meal(s) left')
            else:
                self.chopsticks[first_chopstick].release()


def main():
    philosophers = [0, 1, 2, 3, 4]
    meals_to_eat = 2
    dining_philosophers = DiningPhilosophers(len(philosophers), meals_to_eat)

    with ThreadPoolExecutor(max_workers=5) as pool:
        for philosopher in philosophers:
            pool.submit(dining_philosophers.serveMeals, philosopher)


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

```
Philosopher 2 starts eating...
Philosopher 4 starts eating...
Philosopher 4 stops eating. 1 meal(s) left
Philosopher 0 starts eating...
Philosopher 2 stops eating. 1 meal(s) left
Philosopher 3 starts eating...
Philosopher 3 stops eating. 1 meal(s) left
Philosopher 3 starts eating...
Philosopher 0 stops eating. 1 meal(s) left
Philosopher 3 stops eating. 0 meal(s) left
Philosopher 1 starts eating...
Philosopher 4 starts eating...
Philosopher 1 stops eating. 1 meal(s) left
Philosopher 2 starts eating...
Philosopher 4 stops eating. 0 meal(s) left
Philosopher 2 stops eating. 0 meal(s) left
Philosopher 0 starts eating...
Philosopher 0 stops eating. 0 meal(s) left
Philosopher 1 starts eating...
Philosopher 1 stops eating. 0 meal(s) left
```

- Producer-consumer:

Проблемът производител-потребител е стандартен проблем в компютърните науки, използван за разглеждане на проблеми с нишки или синхронизиране на процеси. Ще разгледате негов вариант, за да получите някои идеи за това какви примитиви предоставя модулът за нишки на Python.

За този пример ще си представите програма, която трябва да чете съобщения от мрежа и да ги запише на диск. Програмата не изисква съобщение, когато пожелае. То трябва да слуша и да приема съобщения, когато идват. Съобщенията няма да идват с редовно темпо, а ще идват на серии. Тази част от програмата се нарича продуцент.

От друга страна, след като имате съобщение, трябва да го напишете в база данни. Достъпът до базата данни е бавен, но достатъчно бърз, за да поддържа средния темп на съобщенията. Той не е достатъчно бърз, за да бъде в крак, когато се появи серия от съобщения. Тази част е потребителят.

Между производителя и потребителя ще създадете конвейер, който ще бъде частта, която се променя, докато научавате за различни обекти за синхронизиране.

Това е основното оформление. Нека разгледаме решение с помощта на Lock. Не работи перфектно, но използва инструменти, които вече познавате, така че е добро място да започнете.

```
import random
from threading import Thread
from queue import Queue
import time


def produce_work(buffer, finished, max_capacity):
    finished.put(False)
    for position in range(max_capacity):
        value = random.randint(1, 100)
        buffer.put(value)
        print(f'producing {value} in position {position}')
    finished.put(True)


def consume_work(buffer, finished):
    position = 0
    while True:
        if buffer.empty():
            print('No items to consume')
            if finished.get() is True:
                break
            continue

        value = buffer.get()
```

```
        print(f'Consuming {value} from position {position}')
        position += 1


def main():
    max_capacity = 20
    buffer = Queue()
    finished = Queue()

    producer = Thread(target=produce_work, args=[buffer, finished, max_capacity])
    consumer = Thread(target=consume_work, args=[buffer, finished])

    producer.start()
    consumer.start()

    producer.join()
    print('Producer finished')

    consumer.join()
    print('Consumer finished')


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

```
producing 7 in position 0
Consuming 7 from position 0
No items to consume
No items to consume
producing 99 in position 1
producing 81 in position 2
producing 74 in position 3
producing 48 in position 4
producing 70 in position 5
producing 24 in position 6
producing 47 in position 7
producing 60 in position 8
producing 44 in position 9
producing 34 in position 10
producing 54 in position 11
producing 15 in position 12
producing 16 in position 13
producing 81 in position 14
producing 24 in position 15
producing 79 in position 16
producing 32 in position 17
producing 11 in position 18
producing 25 in position 19
Producer finished
Consumer finished
```

Част 3 - Заключение

Python поддържа различни подходи свързани с нишковото програмиране, като в този проект бяха разгледани някои основни концепции, както и тяхната същност и имплементация. Програмирането с нишки е изключителмо важно за всяко по-мащабно приложение, особено, ако то обработва, запазва и създава данни. Всеки програмист трябва да знае как да оптимизира създадените от него програми, така че да работят максимално надеждно, бързо и оптимално.

Източници

Книги:

- Mastering Concurrency in Python - Quan Nguyen
- Python Concurrency with asyncio - Matthew Fowler
- Python Cookbook - David Beazley and Brian K. Jones

Курсове:

- https://www.udemy.com/course/the-complete-python-course/
- https://www.udemy.com/course/100-days-of-code/
- https://www.youtube.com/watch?v=YfyQUJ76ktQ

Статии:

- https://betterprogramming.pub/multithreaded-ruby-synchronization-race-conditions-and-deadlocks-f1f1a7cddcea
- https://www.youtube.com/watch?v=YfyQUJ76ktQ