21.     т

18 май 2026

За какво ще си говорим днес?

- Каква е разликата между конкурентност и паралелизъм?
- Кога нишките помагат?
- Кога нишките не помагат?
- Кога други неща помагат?
- Какви са другите неща?
- Какви са проблемите при паралелното програмиране и конкурентността?
- Как двама души могат да акат едновременно в една тоалетна?

Защо ще си говорим днес?

Защото конкурентността е важна тема и защото ако решите да работите на Python има 80% шанс да ви се наложи да направите нещо свързано с нея в следващата една година.

Реални примери - би ни било полезно ако:

- теглим много URL-и
- обработваме много файлове
- имаме background workers
- чакаме база данни / мрежа / API
- имаме CPU-heavy batch задача

Кога ще си говорим днес?

Що е то конкурентност?

~~Когато две изчисления се случват едновременно.~~

Когато две изчисления нямат ясно дефинирана последователност на изпълнение.

... И често (но не задължително) споделят ресурс.

Тоест:

Предварително знаем, че:

- След момент t0 ще започне изпълнението на две задачи T1 и T2
- В момент t1 и двете задачи ще са приключили

Няма как предварително да знаем:

- Реда, в който ще се изпълнят
- Дали ще си предават управлението
- Коя ще приключи първа

Паралелизъм?

Това, за което повечето хора си мислят, когато им говориш за конкурентност.

Две изчисления, които реално се изпълняват едновременно.

Предполага наличието на много ядра/процесори, така че няколко задачи наистина да вървят паралелно.

Пример

Конкурентност (MFF)

Паралелизъм (MFM)

Как се постига конкурентност?

А. Отделни процеси

Б. Нишки

- Зелени нишки - нашия код се грижи за предаването на контрол
- Системни нишки - операционната система се грижи за предаването на контрол

Как се постига конкурентност в Python?

- `threading` - нишки
- `concurrent.futures.ThreadPoolExecutor` - нишки, ама по-лесни за употреба
- `multiprocessing` - процеси
- `concurrent.futures.ProcessPoolExecutor` - процеси, ама по-лесни за употреба
- `asyncio` - следващата лекция

GIL

В стандартния CPython с GIL само една нишка изпълнява Python bytecode в даден момент.

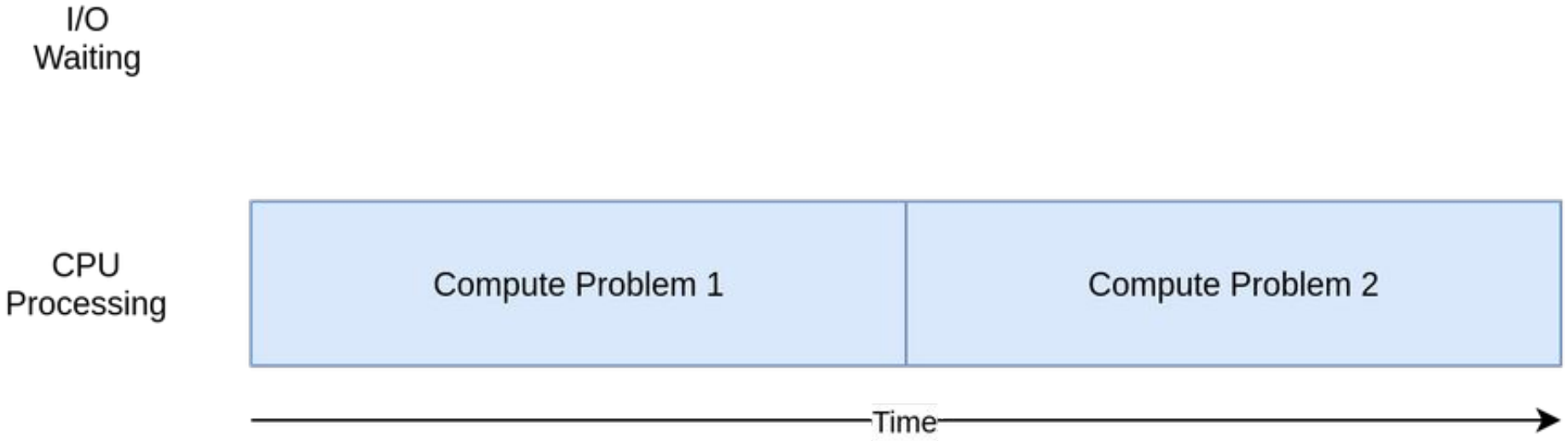
Досадно е, но уви спестява един ред други проблеми.

Възможно е дори референцирането на променлива в няколко нишки едновременно да създаде проблем (защо?).

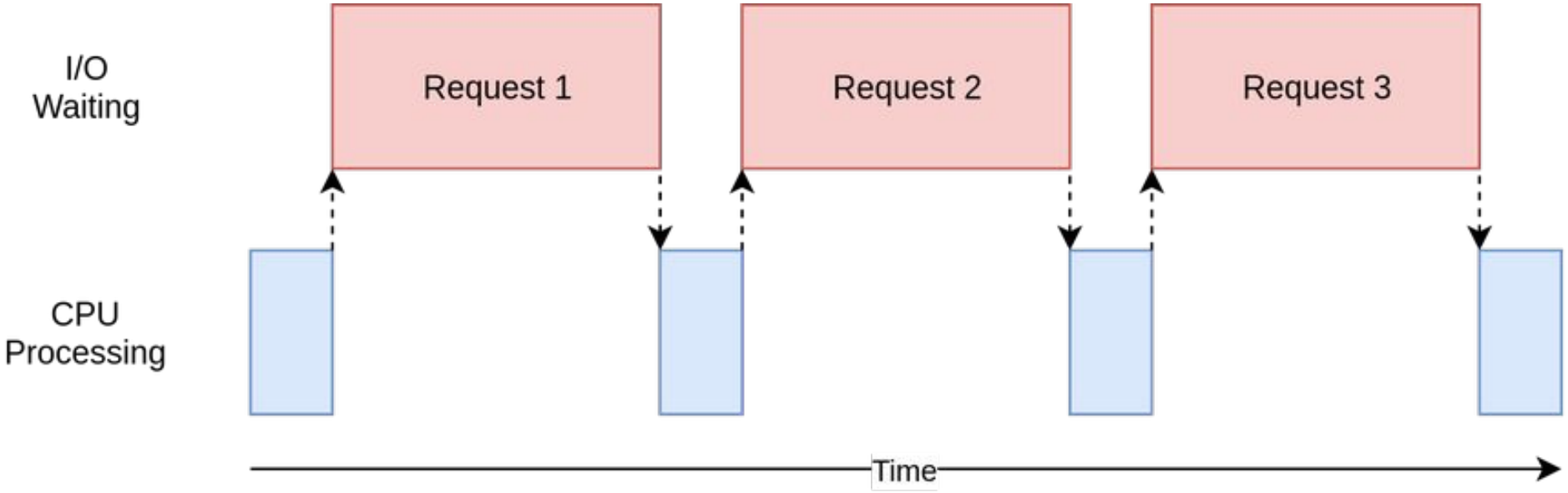
Чакай малко, това звучи супер неефективно!

- Има нещо вярно
- Но!
- Нишките си имат приложение и в Python
- IO-bound vs CPU-bound процеси
- *P.S. От Python 3.13 нататък GIL може да се деактивира*

CPU-bound


I/O
Waiting
CPU
Processing
Compute Problem 1
Compute Problem 2
Time

IO-bound


I/O
Waiting
CPU
Processing
Request 1
Request 2
Request 3
Time

Примери

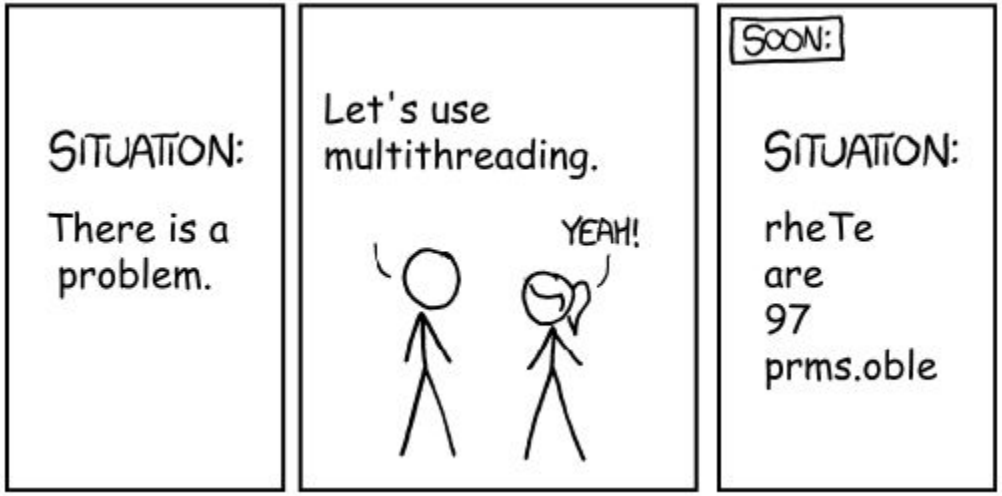
CPU-bound:

- изчисления
- компресиране
- image processing
- тежка логика и алгоритми отвъд горното

IO-bound:

- заявки към мрежа
- заявки към база данни
- четене и писане от / във файлове
- сокети
- изчакване на под-процеси

Да си поговорим за системните нишки



Създаване на нова нишка

Подаваме функция с параметри към нея на `threading.Thread`

или

- Нишка - наследник на `threading.Thread`
- Код - в метода `run`
- Създаване и изпълнение на кода паралелно - метод `start`

Пример с нишки (функция)

```
import threading

def f(name):
    print("Hello from {0}".format(name))

thread = threading.Thread(target=f, args=('Bob',))
thread.start()
thread.join()
```

Пример с нишки 2 (наследник на Thread)

```
import threading
import time

orders = 0

class Chef(threading.Thread):

    def __init__(self, order):
        self.order = order
        threading.Thread.__init__(self)

    def run(self):
        time.sleep(3)
        log("Order '{0}' is ready!".format(self.order))

while True:
    order = input('Enter order: ')
    if not order: continue
    if order in ('q', 'x', 'quit', 'exit'): break
    chef = Chef(order)
    chef.start()
    log("Roger that '{0}'. Please, wait in quiet desperation.".format(order))
    orders += 1
```

Кое от двете е по-добре?

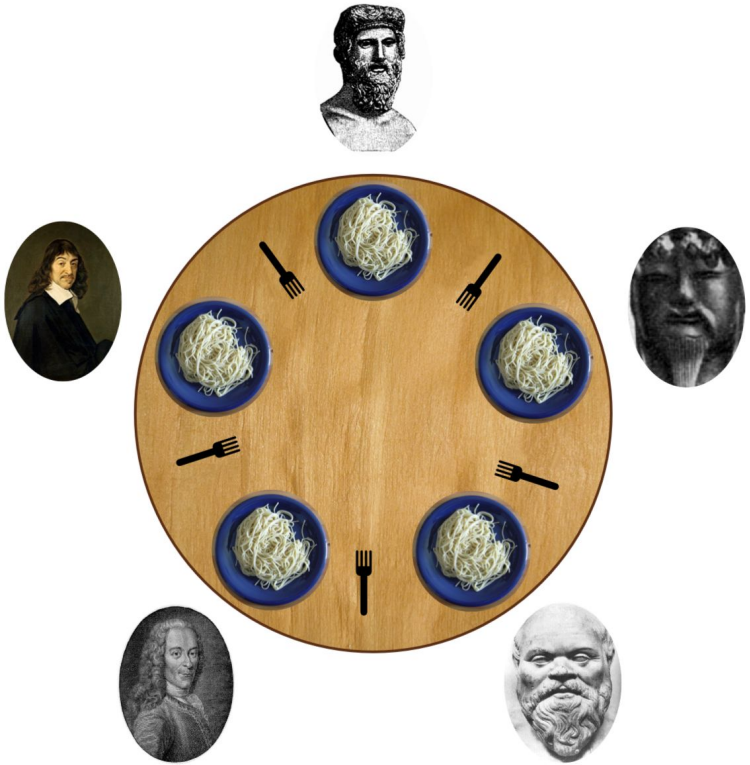
- Между горните две - първото - не искате ненужно complexity
- Но на практика - нито едно от двете
- Ами?
- `ThreadPoolExecutor`
- След малко

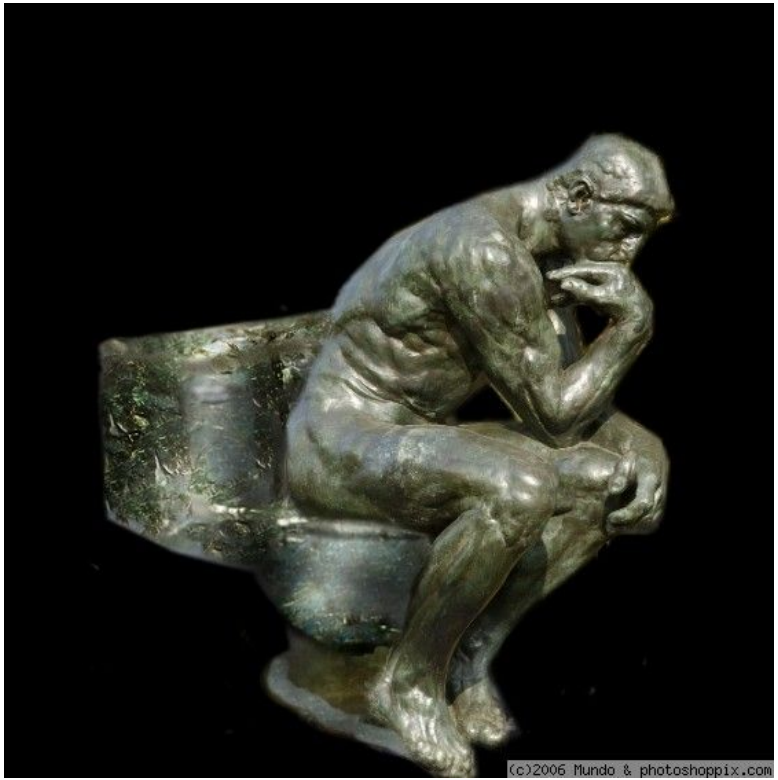
Проблемът с ръчните нишки

Ако имаме 100 задачи:

- кой създава нишките?
- колко нишки са твърде много?
- колко нишки са достатъчно?
- как събираме резултатите?
- как хващаме exceptions?
- как спираме чисто?

Вечерящите философи - проблем!

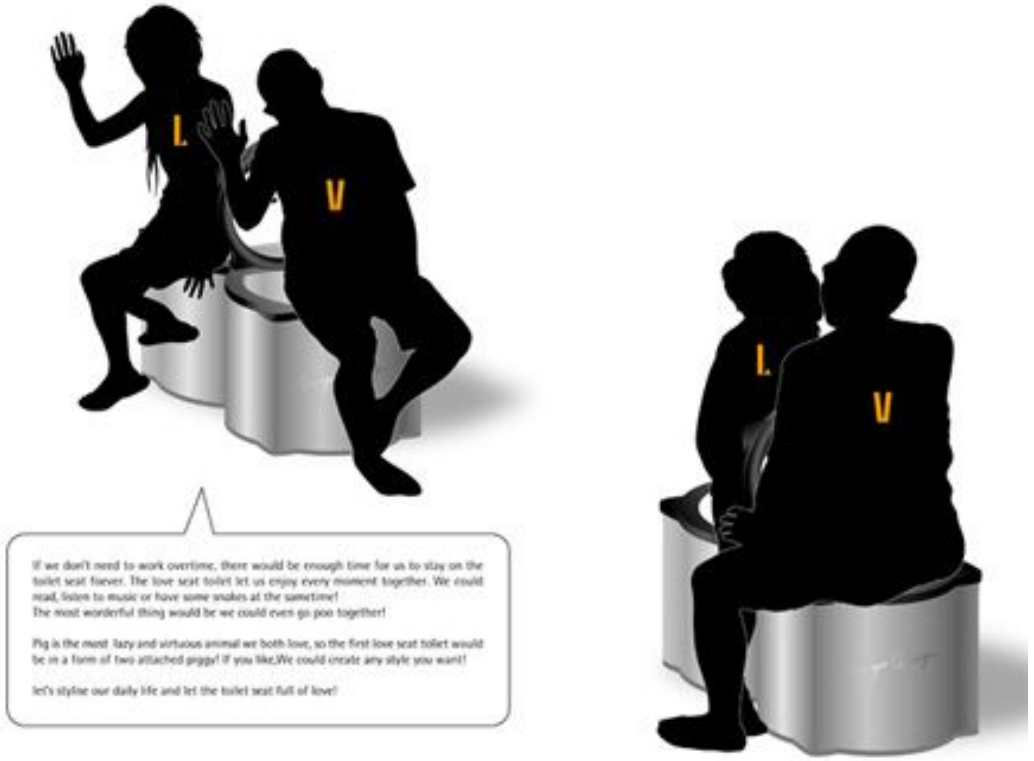




Решение 1

Решение 2

Решение 3


I
V
If we don't need to work overtime, there would be enough time for us to stay on the toilet seat forever. The love seat toilet let us enjoy every moment together. We could read, listen to music or have some snakes at the sametime!
The most wonderful thing would be we could even go poo together!
Pig is the most lazy and virtuous animal we both love, so the first love seat toilet would be in a form of two attached piggy! If you like,We could create any style you want!
let's styline our daily life and let the toilet seat full of love!
I
V

Решение 4

Решение 5

Критични секции!

- Части от кода, които могат да бъдат изпълнени само от една нишка/процес в даден момент се наричат критични секции
- Те са критична част от многозадачното програмиране
- Има много похвати за реализирането на критични секции
- Всичките са равномощни на нещо, наречено **семафор**

Забележка: Кодът от предишният и всички следващи примери, можете да намерите ето тук.

Enter семафори

- От гр. – σήμα (знак) + φορός (носещ)
- Показват на влаковите машинисти дали лентата пред тях е заета или свободна
- Всъщност може да бъде всичко - огън, дим, светлина, знаме
- И да предава всякаква информация - напада ли врага, лошо ли е времето на изток, кои продукти са на промоция в Кауфланд
- Но не точно това имахме предвид

threading.Lock

- `threading.Lock()` ни връща `Lock` обект
- Викането на метода му `acquire()` ни гарантира, че само ние притежаваме този `Lock`
- Викането на `release()` освобождава `Lock`-а и разрешава някой друг да го заключи с `acquire()`
- Ако викнем `acquire()` докато `Lock`-а е зает - методът чака, докато не се освободи

with и обекти с acquire и release

- Всички `Lock`-оподбни обекти от `threading` са и context manager-и
- Ако ги ползваме с `with` ни се гарантира викането на `acquire()` преди и на `release()` след блока

```
with bathroom:
    self.log("--> (entered the bathroom)")
    time.sleep(random.random())
    self.log("<-- (left the bathroom)")
```

Модерно строителство, ресторанти и семафори

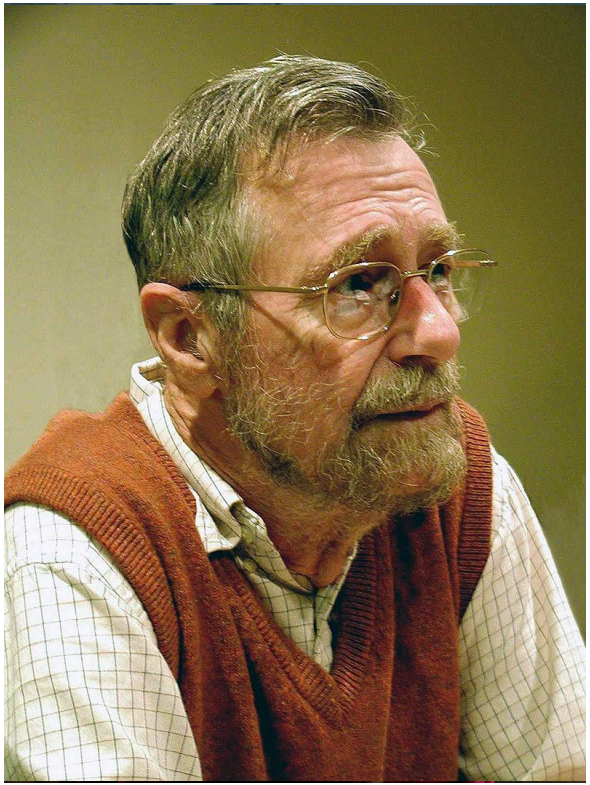
А ако има повече от една тоалетна в сградата?

Или още по-добре:

- Ресторант с 10 човека
- Всеки е едновременно готвач и сервитьор
- 5 фурни

Enter (пак) семафори

- Числова променлива v
- Операция P(v) - чакай докато v > 0, след което v -= 1
- Операция V(v) - v += 1
- Предложени от Дийкстра
- Не този
- Този (Едсхер)



threading.Semaphore

- `threading.Semaphore(k)` ни връща семафор с интерфейс като на `Lock` и стойност `k`
- При всяко изпълнение на `acquire()` стойността се намалява с `1`
- При всяко изпълнение на `release()` стойността се увеличава с `1`
- Ако стойността е `0`, `acquire()` спи, докато някой не я увеличи с `release()`
- `Lock()` е еквивалентен на `Semaphore(1)`

Други threading магии

- Има и други неща като евенти, къндишъни и прочие, все част от **примитивите** на паралелното програмиране.
- Няма да ги обсъждаме, защото всичко дотук е сравнително теоретично.
- Така де, очевидно си има имплементацията в Python, но рядко ни се налага да работим с нишки на такова ниво.

Вместо това...

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

def fetch(url):
    sleep(1)
    return url

urls = ["a", "b", "c", "d", "e"]

print([fetch(url) for url in urls])                    # ~5s
print(list(ThreadPoolExecutor().map(fetch, urls)))     # ~1s
```

fork

Как (unixy) операционна система реализира паралелизъм

- fork създава ново копие на програмата, която изпълняваме
- Всички ресурси и променливи запазват стойността си в процеса-син
- След създаването на новия процес, всички промени са локални
- Все едно клонираме хора, за да вършим повече работа едновременно

Пример с fork (на C/Unix)

```c
#include <stdio.h>

int main()
{
    printf("before\n");
    if (fork())
        printf("father\n");
    else
        printf("son\n");
    printf("both\n");
}
```

```
before
son
both
father
both
```

Пример с fork (на Python)

```
import os
import time

orders = 0

while True:
    order = input('Enter order: ')
    if not order: continue
    if order in ('q', 'x', 'quit', 'exit'): break
    pid = os.fork()
    if pid == 0:
        time.sleep(3)
        print("Order '{0}' is ready!".format(order))
        exit(0)
    else:
        os.wait()
    print("Roger that '{0}'.".format(order))
    orders += 1
```

Предимства и недостатъци на fork

Против:

- Само за UNIX
- Създаването на нов процес е бавно и паметоемко
- Комуникацията между процеси е трудна - нямат обща памет

За:

- Стабилност
- Синът е независим - ако омаже нещо, бащата няма да пострада

multiprocessing

`multiprocessing` модулът

- Дава ни подобни възможности като `threading`, но за процеси
- Интерфейсът е (почти) идентичен с нишка
- Crossplatform - Unix/Windows
- Благинки за синхронизация - `Semaphore`, `Lock`, `RLock`, `Condition`, `Event`
- Благинки за обмен на данни - `Queue`, `Pipe`
- Възможности за обща памет (`Value`, `Array`) от елементарни данни (`int/float/byte/...`) и `ctypes` структури
- Възможност за общи обекти - `Manager`
- Басейн (`Pool`)

Пример с multiprocessing

```
import multiprocessing as mp
import time

def chef_worker(order):
    time.sleep(3)
    print(f"Order '{order}' is ready!")

if __name__ == "__main__":
    orders = 0
    while True:
        order = input("Enter order: ")
        if not order: continue
        if order in ("q", "x", "quit", "exit"): break
        p = mp.Process(target=chef_worker, args=(order,))
        p.start()
        print(f"Roger that '{order}'. Please, wait in quiet desperation.")
        orders += 1
```

multiprocessing - примери

- Пример с обща памет
- Пример с lock
- Нещо много по-удобно - map (ама multiprocessing map)

Паралелизиран map

```
import requests
from multiprocessing import Pool

urls = ['http://www.python.org', 'http://www.python.org/about/',
        'http://github.com/', 'http://www.bg-mamma.com/']

def fetch(url):
        return requests.get(url)

if __name__ == '__main__':
    with Pool(4) as pool:
        statuses = list(map(fetch, urls))
    print(statuses)
```

Неща, за които трябва да се внимава под Windows

- Функциите и аргументите трябва да са „pickle-able“
- Не е хубаво да достъпвате глобални променливи
- Задължително `if __name__ == '__main__'`
- Трябва да се внимава с действията при `import`
- Кодът трябва да живее в модул (т.е. предните примери не работят директно в REPL)

Многонишковото програмиране не е просто

Тук трябва да има слайдове

- ???
- Следващия път.

Напомняне, примерите от лекцията можете да откриете

Тук.

Въпроси?