PPaDS MMXXI

Matúš Jókay, Château de la Mûre
matus.jokay@stuba.sk

uim.fei.stuba.sk/predmet/i-ppds

konzultácie elektronicky

mail, discord

Brad Solomon

Async IO in Python: A Complete Walkthrough

https://realpython.com/async-io-python

https://realpython.com/courses/python-3-concurrency-asyncio-module

Konkurentné programovanie

Konkurentné programovanie

- Konkurencia je jav, keď pohľadom na zdrojový kód nevieme určiť, v akom poradí sa vykonávajú jednotlivé časti kódu

Konkurentné programovanie

- Konkurencia je jav, keď pohľadom na zdrojový kód nevieme určiť, v akom poradí sa vykonávajú jednotlivé časti kódu

- Pri jednovláknovom programe sme to doteraz mali jednoduché: vždy sme uvažovali nad tým, že pri toku vykonávania riadenom zásobníkom nemôže ku konkurencii dochádzať

Konkurentné programovanie

- Konkurencia zavádza možnosť súbežného (paralelného) vykonávania; nie je to však to isté!

Konkurentné programovanie

- Konkurencia zavádza možnosť súbežného (paralelného) vykonávania; nie je to však to isté!

- Pri konkurencii nás zaujíma to, že nevieme poradie vykonaných operácií

Konkurentné programovanie

- Aké možnosti konkurentného vykonávania programov nám prináša úroveň OS?

Konkurentné programovanie

- Aké možnosti konkurentného vykonávania programov nám prináša úroveň OS?

- Procesy
- Vlákna

Konkurentné programovanie

- Procesy

- Prečo konkurencia?

Konkurentné programovanie

- Procesy

- Prečo konkurencia?
- Čo možnosť paralelného vykonávania?

Konkurentné programovanie

- Vlákna

- Prečo konkurencia?

Konkurentné programovanie

- Vlákna

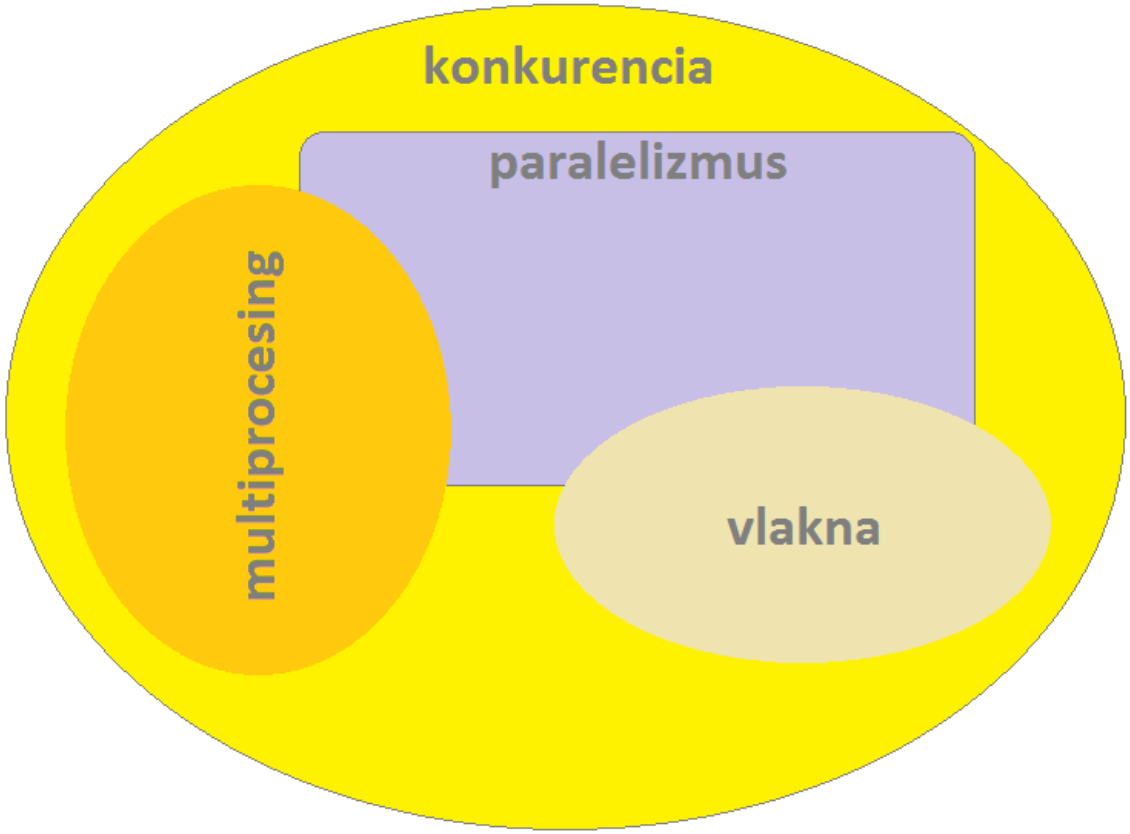
- Prečo konkurencia?
- Čo možnosť paralelného vykonávania?

Konkurentné programovanie

- Zhrnutie

- Konkurencia
- Paralelizmus

- Procesy
- Vlákna



Konkurentné programovanie

- Ako je to v jazyku Python v3.8?

Konkurentné programovanie

- Ako je to v jazyku Python v3.8?
- Nástroje paralelizmu
- Nástroje konkurencie

Konkurentné programovanie

- Ako je to v jazyku Python v3.8?

- Nástroje paralelizmu
  - Procesy

- Nástroje konkurencie
  - Procesy

Konkurentné programovanie

- Ako je to v jazyku Python v3.8?

- Nástroje paralelizmu
  - Procesy

- Nástroje konkurencie
  - Procesy
  - Vlákna
  - Rozšírené generátory, korutiny

Konkurentné programovanie

- Aby sme boli presní, CPython kvôli GIL nepodporuje paralelný beh vlákien takmer vôbec

Konkurentné programovanie

- Aby sme boli presní, CPython kvôli GIL nepodporuje paralelný beh vlákien takmer vôbec

- Iné implementácie (napr. Jython) tento problém nemajú (nemajú totiž GIL) ;)

Konkurentné programovanie

- Dôsledok: možné nepredvídané správanie kódu!

Konkurentné programovanie

- Dôsledok: možné nepredvídané správanie kódu!

- Riešenie: správne riešenie konkurentného prístupu

Konkurentné programovanie

- Dôsledok: možné nepredvídané správanie kódu!
- Riešenie: správne riešenie konkurentného prístupu
- Otázka: stačí riešiť konkurenciu, paralelizmus netreba?

Konkurentné programovanie

- Ako sa dosahuje konkurentnosť?

Konkurentné programovanie

- Ako sa dosahuje konkurentnosť?

- Preemptívny versus kooperatívny multitasking

Konkurentné programovanie

- Ako sa dosahuje konkurentnosť?

- Preemptívny versus kooperatívny multitasking
  - Preemptívny – CPU sa úlohe (task) prideľuje/odoberá transparentne (t.j. úloha o tom "nevie")
  - Kooperatívny – úloha dostane k dispozícii CPU a sama rozhodne, v ktorom okamihu sa ho vzdá

Konkurentné programovanie

- Aj preemptívny, aj kooperatívny multitasking iba **budia dojem** súbežného vykonávania úloh!

Konkurentné programovanie

- Aj preemptívny, aj kooperatívny multitasking iba **budia dojem** súbežného vykonávania úloh!

- Ako?

Konkurentné programovanie

- Aj preemptívny, aj kooperatívny multitasking iba **budia dojem** súbežného vykonávania úloh!

- Ako?
- Vykonávané úlohy (tasky) sa tak rýchlo striedajú, že ľudské zmysly túto výmenu nie sú schopné postrehnúť; naopak, človeku sa javí beh ľubovoľnej aplikácie spojito (napr. video prehrávač)

Konkurentné programovanie

- Preemptívny multitasking sa týka procesov a vlákien

Konkurentné programovanie

- Preemptívny multitasking sa týka procesov a vlákien

- Kooperatívny multitasking bol (veľmi neúspešne) použitý v OS Windows 3.x (95 a 98 pre beh 16-bitových aplikácií)
- V súčasnosti zažíva veľký rozmach nie na úrovni OS, ale na aplikačnej úrovni – async IO

Konkurentné programovanie

- !!! Async IO model programovania nie je založený ani na vláknach, ani na procesoch !!!

Konkurentné programovanie

- !!! Async IO model programovania nie je založený ani na vláknach, ani na procesoch !!!

- Na čom je teda Async IO model založený?

Konkurentné programovanie

- !!! Async IO model programovania nie je založený ani na vláknach, ani na procesoch !!!

- Na čom je teda Async IO model založený?
  - Procesu (vláknu) OS prideľuje CPU na istý čas

Konkurentné programovanie

- !!! Async IO model programovania nie je založený ani na vláknach, ani na procesoch !!!

- Na čom je teda Async IO model založený?
  - Procesu (vláknu) OS prideľuje CPU na istý čas
  - Ak bežiaca úloha na CPU potrebuje čakať na výsledok IO operácie, OS pridelí CPU inej úlohe

Konkurentné programovanie

- !!! Async IO model programovania nie je založený ani na vláknach, ani na procesoch !!!

- Na čom je teda Async IO model založený?
  - Procesu (vláknu) OS prideľuje CPU na istý čas
  - Ak bežiaca úloha na CPU potrebuje čakať na výsledok IO operácie, OS pridelí CPU inej úlohe
  - Prečo však nevyužiť pridelené časové kvantum na vykonávanie iných častí programu?

Konkurentné programovanie

- Async IO paradigma (programovací model) neprišiel na svet s jazykom Python

Konkurentné programovanie

- Async IO paradigma (programovací model) neprišiel na svet s jazykom Python
- Oboznamujeme sa s ním síce pomocou jazyka Python, ale to vďaka tomu, že UŽ je podpora Async IO do jazyka Python integrovaná

Konkurentné programovanie

- Async IO paradigma (programovací model) neprišiel na svet s jazykom Python
- Oboznamujeme sa s ním síce pomocou jazyka Python, ale to vďaka tomu, že UŽ je podpora Async IO do jazyka Python integrovaná

- Viď jazyky Go, C#, Scala, …

Konkurentné programovanie

- Paradigma Async IO nie je založená na vláknach ani multiprocesingu, ale na jedno procesovom (jedno vláknovom) modeli

Konkurentné programovanie

- Paradigma Async IO nie je založená na vláknach ani multiprocesingu, ale na jedno procesovom (jedno vláknovom) modeli

- Využíva kooperatívny multitasking

Konkurentné programovanie

- Paradigma Async IO nie je založená na vláknach ani multiprocesingu, ale na jedno procesovom (jedno vláknovom) modeli

- Využíva kooperatívny multitasking, čím **budí zdanie** súbežného vykonávania úloh

Konkurentné programovanie

- Na DÚ viď prehľad nástrojov konkurentného programovania v Pythone (aj s vysvetlením, kedy sa ktoré hodí a na čo je dobré)

- https://realpython.com/python-concurrency

Async IO

Async IO

- Ako by sme definovali pojem **asynchrónny**?

Async IO

- Ako by sme definovali pojem **asynchrónny**?
  1. Asynchrónne rutiny sú schopné prerušiť svoj beh, pokým neobdržia údaje na pokračovanie v behu. Tým umožnia iným rutinám využiť tento čas na ich beh.

Async IO

- Ako by sme definovali pojem **asynchrónny**?
  1. Asynchrónne rutiny sú schopné prerušiť svoj beh, pokým neobdržia údaje na pokračovanie v behu. Tým umožnia iným rutinám využiť tento čas na ich beh.
  2. Asynchrónny kód, v súlade s bodom 1, umožňuje konkurentné vykonávanie rutín (vyvoláva zdanie ich súbežného vykonávania).

Async IO

- Ako je teda možné dosiahnuť konkurentné vykonávanie, keď máme k dispozícii jedno jadro CPU, jeden (jedno vláknový) proces?

- https://youtu.be/iG6fr81xHKA?t=4m29s

Async IO

- Šachová majsterka Judit Polgar sa predvádza a hrá proti 24 súperom. Má dve možnosti ako hrať: synchrónne alebo asynchrónne

Async IO

- Šachová majsterka Judit Polgar sa predvádza a hrá proti 24 súperom. Má dve možnosti ako hrať: synchrónne alebo asynchrónne
- Čo vieme:
  - 24 protihráčov
  - Judite trvá ťah 5 sekúnd
  - Amatérskemu hráčovi trvá ťah 55 sekúnd
  - Jedna hra má cca 30 ťahov na každej strane (60 obaja hráči spolu)

Async IO

- Synchrónna verzia hry

Async IO

- Synchrónna verzia hry

- Judita hrá jednu hru za druhou
- Jedna hra (55+5)*30 = 1800 sec = 30 min
- 24 hier: 24*30 = 720 min = 12 hod

Async IO

- Asynchrónna verzia hry

Async IO

- Asynchrónna verzia hry

- Judita sa medzi hráčmi hýbe (potiahne a ide k ďalšiemu)
- Judita prejde všetkých 24 stolov za 24*5=120 sec
- 120 > 55 sec, takže protihráč stihne urobiť ťah
- 1 hra 30 ťahov: 120*30 = 3600 sec = 1 hod !!!

Async IO

- Async IO využíva časové intervaly, v ktorých nejaká funkcia (úloha) čaká (je zablokovaná) na beh tých funkcií (úloh), ktoré zablokované nie sú

Async IO

- Async IO využíva časové intervaly, v ktorých nejaká funkcia (úloha) čaká (je zablokovaná) na beh tých funkcií (úloh), ktoré zablokované nie sú

- Poznámka: funkcia, ktorá je zablokovaná čakaním na dokončenie IO v systémovom volaní, znemožňuje vykonávanie akejkoľvek inej časti kódu procesu (!!! jednovláknový proces !!!)

Async IO versus vlákna

Async IO versus vlákna

- V čom je výhoda Async IO oproti programovaciemu modelu založenému na vláknach?

Async IO versus vlákna

- V čom je výhoda Async IO oproti programovaciemu modelu založenému na vláknach?

- Vlákna zdieľajú adresný priestor!!! Zdieľajú spoločnú pamäť procesu!!!

Async IO versus vlákna

- Pri programovaní vlákien treba dbať na konkurentný prístup ku zdieľaným údajovým štruktúram

Async IO versus vlákna

- Pri programovaní vlákien treba dbať na konkurentný prístup ku zdieľaným údajovým štruktúram

- Tejto problematike sme sa venovali v prvej časti semestra: aký synchronizačný mechanizmus sme používali na ochranu integrity pamäťového miesta pri súčasnom (modify/write) prístupe viacerých vlákien?

Async IO versus vlákna

- Tento problém pri Async IO odpadá ÚPLNE a CELKOM

Async IO versus vlákna

- Tento problém pri Async IO odpadá ÚPLNE a CELKOM

- Prečo? Pretože máme jednovláknovú aplikáciu, takže v jednom čase sa môže vykonávať iba jedna jediná inštrukcia programu

Async IO versus vlákna

- To však neznamená, že Async IO je lepšie riešenie než vlákna

Async IO versus vlákna

- To však neznamená, že Async IO je lepšie riešenie než vlákna

- Ani to nie je pravda, že tvorba aplikácie, ktorá je plne Async IO, je jednoduché a ľahké

Async IO versus vlákna

- To však neznamená, že Async IO je lepšie riešenie než vlákna

- Ani to nie je pravda, že tvorba aplikácie, ktorá je plne Async IO, je jednoduché a ľahké
- Asyncio v Pythone je založené na entitách ako callbacks, events, transports, protocols, futures; znie to azda jednoducho?

async/await a generátory

async/await a generátory

- Python vo verzii 3.5 zaviedol nové kľúčové slová `async`/`await`, o ktorých ešte bude reč neskôr

- Dovtedy nejestvovala v jazyku Python podpora tzv. natívnych korutín; korutiny boli implementované pomocou rozšírených generátorov

async/await a generátory

- Ako?

async/await a generátory

- Ako?

```
>>> def py34_coro_ver_a():
        for i in stuff():
            yield i


>>> def py34_coro_ver_b():
        yield from stuff()
```

async/await a generátory

- Podpora korutín založených na generátoroch bude z jazyka Python odstránená vo verzii 3.10

- Presnejšie, týka sa to dekorátora `@asyncio.coroutine`
- https://docs.python.org/3/library/asyncio-task.html#generator-based-coroutines

async/await a generátory

- Rozdiel medzi funkciou a generátorom

- Funkcia
  - Všetko alebo nič
  - Keď sa raz začne vykonávať, ukončí sa až príkazom return
- Generátor
  - Preruší svoj beh pri dosiahnutí riadku yield
  - Dokáže dodať hodnotu a obnoviť svoj beh spolu so stavom lokálnych premenných

async/await a generátory

- Korutiny založené na generátoroch sme využili na pochopenie problematiky
- V ďalšom texte sa budeme venovať výlučne novej syntaxi jazyka, ktorá podporuje natívne korutiny

async/await a natívne korutiny

async/await a natívne korutiny

- Zopakujme, čo je korutina

async/await a natívne korutiny

- Zopakujme, čo je korutina:

- Funkcia, ktorá dokáže pozastaviť svoje vykonávanie predtým, než dosiahne koniec (pomocou príkazu return), a (nepriamo) dokáže na istý čas odovzdať vykonávanie inej korutine

async/await a natívne korutiny

- Skúsme porovnať príklad synchrónnej a asynchrónnej verzie krátkeho ilustračného programu

```python
#!/usr/bin/env python3
# countsync.py

import time

def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    for _ in range(3):
        count()

if __name__ == "__main__":
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```python
#!/usr/bin/env python3
# countsync.py

import time

def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    for _ in range(3):
        count()

if __name__ == "__main__":
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```
$ python3 countsync.py
One
Two
One
Two
One
Two
countsync.py executed in 3.01 seconds.
```

```python
#!/usr/bin/env python3
# countasync.py

import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```python
#!/usr/bin/env python3
# countasync.py

import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```
$ python3 countasync.py
One
One
One
Two
Two
Two
countasync.py executed in 1.01 seconds.
```

```python
#!/usr/bin/env python3
# countasync.py

import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

async/await a natívne korutiny

- Synchrónna verzia

async/await a natívne korutiny

- Synchrónna verzia
  - Vykonáva sekvenčne 3x funkciu count()
  - sleep() vo funkcii count() ilustruje dobu vykonania nejakej IO operácie

async/await a natívne korutiny

- Synchrónna verzia
  - Vykonáva sekvenčne 3x funkciu count()
  - sleep() vo funkcii count() ilustruje dobu vykonania nejakej IO operácie
- Asynchrónna verzia

async/await a natívne korutiny

- Synchrónna verzia
  - Vykonáva sekvenčne 3x funkciu count()
  - sleep() vo funkcii count() ilustruje dobu vykonania nejakej IO operácie
- Asynchrónna verzia
  - sleep() vo funkcii count() nevykoná zablokovanie funkcie, ale pozastaví jej vykonávanie, pokým výsledok (tu vypršanie doby čakania) nebude k dispozícii a riadenie vráti tzv. “slučke udalostí”

async/await a natívne korutiny

- Synchrónna verzia
  - Vykonáva sekvenčne 3x funkciu count()
  - sleep() vo funkcii count() ilustruje dobu vykonania nejakej IO operácie
- Asynchrónna verzia
  - sleep() vo funkcii count() nevykoná zablokovanie funkcie, ale pozastaví jej vykonávanie, pokým výsledok (tu vypršanie doby čakania) nebude k dispozícii a riadenie vráti tzv. “slučke udalostí”
  - Slučka udalostí vyberá na beh funkciu, ktorá môže bežať

async/await a natívne korutiny

- Keď úloha príde k riadku “`await asyncio.sleep(1)`”, riadenie sa vráti slučke udalostí

async/await a natívne korutiny

- Keď úloha príde k riadku “`await asyncio.sleep(1)`”, riadenie sa vráti slučke udalostí
- “Niekde” na pozadí sa poznamená, že túto úlohu je potrebné znovu “prebudiť” o 1 sekundu

async/await a natívne korutiny

- Keď úloha príde k riadku “`await asyncio.sleep(1)`”, riadenie sa vráti slučke udalostí
- “Niekde” na pozadí sa poznamená, že túto úlohu je potrebné znovu “prebudiť” o 1 sekundu
- Medzitým sa môže vykonávať iná úloha

async/await a natívne korutiny

- Rozdiel verzií vidno na výstupoch

async/await a natívne korutiny

- Rozdiel verzií vidno na výstupoch:
  - Dĺžka behu
  - Výpisy

async/await a natívne korutiny

- Rozdiel verzií vidno na výstupoch:
  - Dĺžka behu
  - Výpisy

- Aj keď máme jedno vlákno, dosiahli sme zdanie súčasného behu viacerých funkcií naraz!

async/await a natívne korutiny

- Syntax `async def` určuje
  - Natívnu korutinu
  - Asynchrónny generátor

async/await a natívne korutiny

- Syntax `async def` určuje
  - Natívnu korutinu
  - Asynchrónny generátor

- Výrazy `async with` a `async for` sú taktiež platné; vysvetlenie príde neskôr

async/await a natívne korutiny

- Kľúčove slovo `await` odovzdáva riadenie späť do cyklu spracovania udalostí (*event loop*, slučka udalostí)

- Dôsledok: vykonávanie kódu sa preruší

async/await a natívne korutiny

```
async def g():
    # Pause here and come back to g() when f() is ready
    r = await f()
    return r
```

- Vo funkcii `g()` nech je riadok `await f()`
- Riadenie sa vráti do cyklu spracovania udalostí spolu s informáciou, že beh funkcie `g()` bude prerušený, pokým nebude k dispozícii výsledok volania funkcie `f()`
- Slučka udalostí môže medzitým nechať bežať inú úlohu (obnoviť beh nejakej, alebo nejakú spustiť)

async/await a natívne korutiny

- Pravidlá kedy a ako (ne)používať async/await

async/await a natívne korutiny

- Pravidlá kedy a ako (ne)používať async/await

1. Definícia funkcie, pred ktorou sa nachádza `async`, je definíciou korutiny

2. Ako je (syntaktickou) chybou použiť kľúčové slovo `yield` mimo `def` bloku, tak je chybou použiť `await` mimo `async def` bloku

async/await a natívne korutiny

- Blok `async def`
  - Povolené `await`, `return`, `yield`
  - Všetky tri voliteľne
  - Platná definícia: `async def noop(): pass`

- Nie je možné priamo zavolať takto definovanú korutinu, aby vrátila nejaký výsledok!

async/await a natívne korutiny

```
>>> async def noop(): pass

>>> noop()
<coroutine object noop at 0x00000000038F90C0>
>>>
>>> await noop()
SyntaxError: 'await' outside function
>>>
```

async/await a natívne korutiny

```
>>> def run_noop():
        print(noop())


>>> run_noop()
<coroutine object noop at 0x00000000038F9140>

Warning (from warnings module):
  File "<pyshell#68>", line 2
RuntimeWarning: coroutine 'noop' was never awaited
>>>
```

async/await a natívne korutiny

```
>>> def run_noop():
        await noop()

SyntaxError: 'await' outside async function
>>>
```

async/await a natívne korutiny

```
>>> async def run_noop():
        await noop()


>>>
>>> run_noop()
<coroutine object run_noop at 0x00000000038F9140>
>>>
```

async/await a natívne korutiny

```
>>> import asyncio as aio
>>>
>>> aio.run(noop)
Traceback (most recent call last):
  File "<pyshell#141>", line 1, in <module>
    aio.run(noop)
  File "C:\Program Files\Python38\lib\asyncio\runn
ers.py", line 37, in run
    raise ValueError("a coroutine was expected, go
t {!r}".format(main))
ValueError: a coroutine was expected, got <functio
n noop at 0x00000000038FA0D0>
>>>
```

async/await a natívne korutiny

```
>>>
>>> import asyncio as aio
>>>
>>> aio.run(noop())
>>>
```

async/await a natívne korutiny

- Keď píšeme vo funkcii riadok `await f()`, znamená to, že funkcia `f()` musí byť čakateľná (*awaitable*)

async/await a natívne korutiny

- Keď píšeme vo funkcii riadok `await f()`, znamená to, že funkcia `f()` musí byť čakateľná (*awaitable*)

- Buď ide tiež o korutinu (`async def xyz()`)
- Alebo o objekt, ktorý implementuje metódu `__await()__`, ktorá vracia iterátor

async/await a natívne korutiny

- Ktoré objekty su čakateľné?

- https://docs.python.org/3/reference/datamodel.html#awaitable-objects

async/await príklad

async/await príklad

- Majme program, ktorý 3x volá úlohu `makerandom()`. Táto úloha generuje náhodné čísla v rozsahu <0;10>, pokým vygenerované číslo nepresiahne hranicu určenú argumentom úlohy

async/await príklad

- Majme program, ktorý 3x volá úlohu `makerandom()`. Táto úloha generuje náhodné čísla v rozsahu <0;10>, pokým vygenerované číslo nepresiahne hranicu určenú argumentom úlohy
- Cieľom je vytvoriť takú verziu programu, aby mohli konkurentne bežať viaceré inštancie úlohy (teda nie sériovo za sebou)!

async/await príklad

```
import random
import time


def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")


def main():
    for i in range(3):
        makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    main()
```

```python
import random
import time


def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


def main():
    for i in range(3):
        makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    main()
    print(f"\nTime elapsed:{time.time()-t}")
```

```
Initiated makerandom(0).
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 0 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 7 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 8 too low; retrying.
---> Finished: makerandom(0) == 10
Initiated makerandom(1).
makerandom(1) == 7 too low; retrying.
makerandom(1) == 8 too low; retrying.
makerandom(1) == 4 too low; retrying.
makerandom(1) == 7 too low; retrying.
makerandom(1) == 1 too low; retrying.
makerandom(1) == 6 too low; retrying.
---> Finished: makerandom(1) == 9
Initiated makerandom(2).
makerandom(2) == 3 too low; retrying.
---> Finished: makerandom(2) == 9

Time elapsed:23.11232590675354
>>>
```

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    for i in range(3):
        await makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

async/await príklad

- Stačí takáto zmena, aby sa zo sériového vykonávania stalo konkurentné pomocou asyncio?

async/await príklad

- Stačí takáto zmena, aby sa zo sériového vykonávania stalo konkurentné pomocou asyncio?

- Pozrime si výstup!

```
Initiated makerandom(0).
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 0 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 7 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 8 too low; retrying.
---> Finished: makerandom(0) == 10
Initiated makerandom(1).
makerandom(1) == 7 too low; retrying.
makerandom(1) == 8 too low; retrying.
makerandom(1) == 4 too low; retrying.
makerandom(1) == 7 too low; retrying.
makerandom(1) == 1 too low; retrying.
makerandom(1) == 6 too low; retrying.
---> Finished: makerandom(1) == 9
Initiated makerandom(2).
makerandom(2) == 3 too low; retrying.
---> Finished: makerandom(2) == 9

Time elapsed: 23.070329427719116
>>>
```

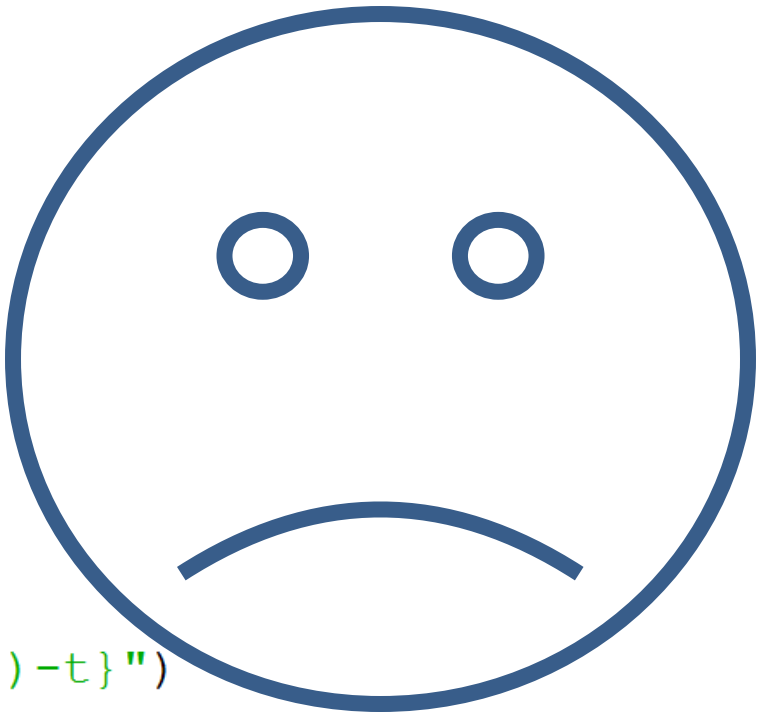
async/await príklad

- V čom je problém, keď máme definované všetky potrebné náležitosti asynchrónnej aplikácie?

- Slučka udalostí (`asyncio.run()`)
- Asynchrónne funkcie (`async`)
- Asynchrónne volanie funkcií (`await`)

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    for i in range(3):
        await makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    for i in range(3):
        await makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```



async/await príklad

- Aj keď je funkcia `main`() definovaná ako asynchrónna a využíva asynchrónne volanie ďalšej korutiny, tak jednotlivé volania korutiny sú sériovo v cykle!

- Pomocou `asyncio.run`() sme spustili asynchrónnu funkciu `main`(), ale korutiny spúšťame sériovo, nie konkurentne

async/await príklad

- V podstate môžeme využiť dve funkcie modulu `asyncio`, ktoré implementujú konkurentné spúšťanie korutín, ktoré sú zadané ako argumenty

async/await príklad

- V podstate môžeme využiť dve funkcie modulu `asyncio`, ktoré implementujú konkurentné spúšťanie korutín, ktoré sú zadané ako argumenty

- `asyncio.gather()` a `asyncio.wait()`
- https://stackoverflow.com/questions/42231161/asyncio-gather-vs-asyncio-wait

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    await asyncio.gather(*(makerandom(i, 10-i-1) for i in range(3)))


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

```
Initiated makerandom(0).
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 0 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 7 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 8 too low; retrying.
---> Finished: makerandom(0) == 10
Initiated makerandom(1).
makerandom(1) == 7 too low; retrying.
makerandom(1) == 8 too low; retrying.
makerandom(1) == 4 too low; retrying.
makerandom(1) == 7 too low; retrying.
makerandom(1) == 1 too low; retrying.
makerandom(1) == 6 too low; retrying.
---> Finished: makerandom(1) == 9
Initiated makerandom(2).
makerandom(2) == 3 too low; retrying.
---> Finished: makerandom(2) == 9

Time elapsed:23.063318967819214
>>>
```

async/await príklad

- Blokujúce volanie vo funkcii `makerandom`!!!

async/await príklad

- Blokujúce volanie vo funkcii `makerandom`!!!

- Dôsledky / príčiny
  - Vlákno procesu musí čakať, zablokuje sa vykonávanie
  - Nemôže sa odovzdať riadenie slučke udalostí
  - Nikde vo funkcii `makerandom`() nie je `await`!

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        #time.sleep(idx+1)
        await asyncio.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    await asyncio.gather(*(makerandom(i, 10-i-1) for i in range(3)))


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

```
Initiated makerandom(0).
makerandom(0) == 4 too low; retrying.
Initiated makerandom(1).
makerandom(1) == 4 too low; retrying.
Initiated makerandom(2).
makerandom(2) == 0 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(1) == 7 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(2) == 4 too low; retrying.
makerandom(0) == 8 too low; retrying.
---> Finished: makerandom(1) == 10
makerandom(0) == 7 too low; retrying.
makerandom(0) == 8 too low; retrying.
makerandom(2) == 4 too low; retrying.
makerandom(0) == 7 too low; retrying.
makerandom(0) == 1 too low; retrying.
makerandom(0) == 6 too low; retrying.
---> Finished: makerandom(2) == 9
makerandom(0) == 3 too low; retrying.
makerandom(0) == 9 too low; retrying.
makerandom(0) == 7 too low; retrying.
---> Finished: makerandom(0) == 10

Time elapsed:12.163782835006714
>>>
```

async/await príklad

- Poznámky ku príkladu

- Príklad generovania náhodných čísel nie je veľmi vhodný pre `asyncio`…
- IO operácia je modelovaná pomocou `sleep`!
- Rôzne doby trvania IO operácie sú modelované pomocou argumentu `sleep`()

Async IO návrhové vzory

Async IO návrhové vzory

1. Zreťazenie korutín

2. Producenti-konzumenti

Async IO návrhové vzory

- Zreťazenie je (syntakticky) triviálne
- Netriviálny je logický návrh programu ;)

- Viď ukážka chained.py

```python
async def part1(n: int) -> str:
    i = random.randint(0, 10)
    print(f"part1({n}) sleeping for {i} seconds.")
    await asyncio.sleep(i)
    result = f"--> res{n}-1"
    print(f"Returning part1({n}) == '{result}'.")
    return result

async def part2(n: int, arg: str) -> str:
    i = random.randint(0, 10)
    print(f"part2{n, arg} sleeping for {i} seconds.")
    await asyncio.sleep(i)
    result = f"{arg} --> res{n}-2"
    print(f"Returning part2{n, arg} == '{result}'.")
    return result

async def chain(n: int) -> None:
    p1 = await part1(n)
    p2 = await part2(n, p1)
    print(f"*** Chained res{n} => '{p2}'.")

async def main(*args):
    await asyncio.gather(*(chain(n) for n in args))

if __name__ == "__main__":
    random.seed(444)
    asyncio.run(main(1,2,3))
    print(f"Program finished.")
```

```
part1(1) sleeping for 4 seconds.
part1(2) sleeping for 4 seconds.
part1(3) sleeping for 0 seconds.
Returning part1(3) == '--> res3-1'.
part2(3, '--> res3-1') sleeping for 4 seconds.
Returning part1(1) == '--> res1-1'.
part2(1, '--> res1-1') sleeping for 7 seconds.
Returning part1(2) == '--> res2-1'.
part2(2, '--> res2-1') sleeping for 4 seconds.
Returning part2(3, '--> res3-1') == '--> res3-1 --> res3-2'.
*** Chained res3 => '--> res3-1 --> res3-2'.
Returning part2(2, '--> res2-1') == '--> res2-1 --> res2-2'.
*** Chained res2 => '--> res2-1 --> res2-2'.
Returning part2(1, '--> res1-1') == '--> res1-1 --> res1-2'.
*** Chained res1 => '--> res1-1 --> res1-2'.
Program finished.
>>>
```

Async IO návrhové vzory

- Na príklade vidíme, že `part2()` sa začne vykonávať až po ukončení korutiny `part1()`
- Toto správanie sme tu už dnes raz mali…

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    for i in range(3):
        await makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

Async IO návrhové vzory

- Na príklade vidíme, že `part2()` sa začne vykonávať až po ukončení korutiny `part1()`
- Toto správanie sme tu už dnes raz mali…
- V onom prípade to bolo nežiadúce správanie, pretože sme chceli korutiny vykonávať konkurentne
- Niekedy však môže byť vykonávanie korutín od seba závislé

```
part1(1) sleeping for 4 seconds.
part1(2) sleeping for 4 seconds.
part1(3) sleeping for 0 seconds.
Returning part1(3) == 'res3-1'.
part2(3, 'res3-1') sleeping for 4 seconds.
Returning part1(1) == 'res1-1'.
part2(1, 'res1-1') sleeping for 7 seconds.
Returning part1(2) == 'res2-1'.
part2(2, 'res2-1') sleeping for 4 seconds.
Returning part2(3, 'res3-1') == 'res3-2 from res3-1'.
-->Chained res3 => 'res3-2 from res3-1' (took 4.05 seconds).
Returning part2(2, 'res2-1') == 'res2-2 from res2-1'.
-->Chained res2 => 'res2-2 from res2-1' (took 8.06 seconds).
Returning part2(1, 'res1-1') == 'res1-2 from res1-1'.
-->Chained res1 => 'res1-2 from res1-1' (took 11.12 seconds).
Program finished in 11.12 seconds.
>>>
```

Async IO návrhové vzory

- Druhý vzor programovania Async IO uvedený v tejto prednáške je model P-K

Async IO návrhové vzory

- Druhý vzor programovania Async IO uvedený v tejto prednáške je model P-K

- PK problém vyžaduje “sklad”
- Python poskytuje modul `queue` (aj) na tieto účely (“odolný” aj v prípade použitia vlákien)
- https://docs.python.org/3/library/queue.html#module-queue

Async IO návrhové vzory

- `asyncio` modul poskytuje podobnú funkcionalitu, určenú pre asyncio použitie

- https://docs.python.org/3/library/asyncio-queue.html

Async IO návrhové vzory

- Kedy použiť PK, kedy serializáciu menších/kratších korutín?

Async IO návrhové vzory

- Kedy použiť PK, kedy serializáciu menších/kratších korutín?

- V prípade PK konzumenti nepoznajú ani počet producentov, ani počet položiek, ktoré treba spracovať
- P ani K navzájom priamo nekomunikujú, iba sprostredkovane cez položky fronty

Async IO návrhové vzory

- Synchrónna verzia riešenia PK je nepoužiteľná

- Istý počet producentov sériovo vkladá položky do fronty
- Až keď všetci producenti ukončia vkladanie, môžu začať konzumenti vyberať položky z fronty a spracúvať ich. Taktiež sériovo, položku po položke.

Async IO návrhové vzory

- Pri modeli PK môže byť problémom notifikácia konzumentov o tom, že už žiadne položky do fronty nepribudnú

- Problémom je, aby konzumenti neostali navždy čakať v metóde `q.get()`

Async IO návrhové vzory

- Riešenie (hlavnej, koordinujúcej úlohy):
  1. Vytvor frontu
  2. Vytvor úlohy producentov
  3. Vytvor úlohy konzumentov
  4. Počkaj na dokončenie producentov
  5. Počkaj na vyprázdnenie fronty
  6. Zruš úlohy konzumentov

Async IO návrhové vzory

```
async def produce(n: int, q: asyncio.Queue) -> None:
    # Synchronous loop for each single producer
    for _ in range(n):
        item = await makeitem()
        await q.put(item)

async def consume(q: asyncio.Queue) -> None:
    while True:
        item = await q.get()
        await consume_item(item)
        q.task_done()

async def main(nprod: int, ncon: int) -> None:
    q = asyncio.Queue()
    producers = [asyncio.create_task(produce(n, q)) for n in range(nprod)]
    consumers = [asyncio.create_task(consume(n, q)) for n in range(ncon)]
    await asyncio.gather(*producers)
    await q.join()  # Implicitly awaits consumers, too
    for c in consumers:
        c.cancel()

if __name__ == "__main__":
    asyncio.run(main(3,10))
```

Cyklus spracovania udalostí

Cyklus spracovania udalostí

- Angl. *event loop*

- Predstavme si pre jednoduchosť cyklus `while True`

Cyklus spracovania udalostí

- Angl. *event loop*

- Predstavme si pre jednoduchosť cyklus `while True`, ktorý
  - Monitoruje, ktoré úlohy sú nečinné
  - Zisťuje, ktorá úloha môže byť vykonávaná, kým iná sa stala nečinnou (pozastavenou)
  - Obnovuje beh úlohy, pre ktorú je k dispozícii udalosť, na ktorú čakala

Cyklus spracovania udalostí

- Celý tento menežment ohľadom úloh sa dá v Pythone (od 3.7) urobiť jediným riadkom!

```
asyncio.run(main())
```

Cyklus spracovania udalostí

- Celý tento menežment ohľadom úloh sa dá v Pythone (od 3.7) urobiť jediným riadkom!

```
asyncio.run(main())
```

- Tento riadok je zodpovedný za to, že
  - Sa získa inštancia cyklu spracovania udalostí
  - Úlohy sa budú spúšťať, pokým nebudú označené ako ukončené
  - Cyklus spracovania udalostí sa korektne ukončí

Cyklus spracovania udalostí

- Pokým nebola funkcia run() k dispozícii (alebo ak potrebujeme väčšiu kontrolu nad cyklom udalostí), využiva(l) sa nasledovný kód

Cyklus spracovania udalostí

- Pokým nebola funkcia run() k dispozícii (alebo ak potrebujeme väčšiu kontrolu nad cyklom udalostí), využiva(l) sa nasledovný kód

```
loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

Cyklus spracovania udalostí

- Vždy treba nejako začať vykonávať program
- Pri asynchrónnom programovaní platí, že väčšinou dávame funkcii `run`() ako parameter hlavnú asynchrónnu funkciu, ktorá podľa potreby volá ďalšie

Cyklus spracovania udalostí

- Vždy treba nejako začať vykonávať program
- Pri asynchrónnom programovaní platí, že väčšinou dávame funkcii `run`() ako parameter hlavnú asynchrónnu funkciu, ktorá podľa potreby volá ďalšie

- V predošlej časti sme už ukazovali, že asynchrónnu funkciu nevieme spustiť klasickým spôsobom…

Konkurencia v Pythone

Konkurencia v Pythone

- Kedy zvoliť *multiprocessing*, kedy *threading*, kedy *asyncio*?

- Môžu koexistovať v jednom programe?
- Aké kombinácie “sú povolené”?

Konkurencia v Pythone

- Keď máme dobre škálovateľnú úlohu na nezávislé spracovanie na jednotlivých výpočtových uzloch, a tieto uzly máme k dispozícii, jednoznačne **multiprocessing**

Konkurencia v Pythone

- Keď máme dobre škálovateľnú úlohu na nezávislé spracovanie na jednotlivých výpočtových uzloch, a tieto uzly máme k dispozícii, jednoznačne **multiprocessing**
- Ak máme vyvíjať aplikáciu, ktorá veľa času trávi spracovaním IO (sieťová komunikácia, práca s externými pamäťami, komunikácia s používateľom), jednoznačne **asyncio**

Konkurencia v Pythone

- Keď máme dobre škálovateľnú úlohu na nezávislé spracovanie na jednotlivých výpočtových uzloch, a tieto uzly máme k dispozícii, jednoznačne **multiprocessing**
- Ak máme vyvíjať aplikáciu, ktorá veľa času trávi spracovaním IO (sieťová komunikácia, práca s externými pamäťami, komunikácia s používateľom), jednoznačne **asyncio**
- Ostatné môže byť **threading**

Konkurencia v Pythone

- Treba mať na pamäti, že najťažšie sa ladia viacvláknové aplikácie
- Zvlášť preto, lebo často nie je možné zopakovať sled operácií tak, aby sa chyba dala zreprodukovať – veľká miera náhodnosti

Konkurencia v Pythone

- Treba mať na pamäti, že najťažšie sa ladia viacvláknové aplikácie
- Zvlášť preto, lebo často nie je možné zopakovať sled operácií tak, aby sa chyba dala zreprodukovať – veľká miera náhodnosti

- Avšak asynchrónna aplikácia nie je záchranou a liekom na všetko! V prípade úloh, ktoré málo interagujú s IO, môže byť výsledná aplikácia nielen neprehľadná, ale aj pomalá

Konkurencia v Pythone

- Modul `asyncio` nie je jediným, ktorý v Pythone dokáže menežovať korutiny

- https://github.com/dabeaz/curio
- https://github.com/python-trio/trio
- …

Cvičenie

- Prepracovanie synchrónnej aplikácie na asynchrónnu
- Využitie asynchrónneho programovania pri spracované HTTP žiadostí
- Porovnanie efektivity synchrónneho versus asynchrónneho prístupu

Zdroje

- Krátky, jasný a výstižný úvod do korutín: https://pymotw.com/3/asyncio/coroutines.html

- Vynikajúci prehľad vývoja asynchrónneho programovania v Pythone po 3.5: https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/

Zdroje

- Prednáška bola robená podľa: https://realpython.com/async-io-python/

- Načo je dobré Async IO? Napríklad milión http requestov za 9 minút: https://pawelmhm.github.io/asyncio/python/aiohttp/2016/04/22/asyncio-aiohttp.html

Zdroje

- Pekne udržiavaný dôkladný a prehľadný popis AsyncIO: https://yeray.dev/python/asyncio/asyncio-for-the-working-python-developer

PEP

- PEP 342 – Coroutines via Enhanced Generators
- PEP 380 – Syntax for Delegating to a Subgenerator
- PEP 3153 – Asynchronous IO support
- PEP 3156 – Asynchronous IO Support Rebooted: the “asyncio” Module
- PEP 492 – Coroutines with async and await syntax
- PEP 525 – Asynchronous Generators
- PEP 530 – Asynchronous Comprehensions