22. Асинхронно програмиране

20 май 2026

Слайд за конкурентност

Какво ще получим от следния код?

```
counter = 0

def increment_counter():
    global counter
    for _ in range(1000000):
        counter += 1

thread1 = threading.Thread(target=increment_counter)
thread2 = threading.Thread(target=increment_counter)
thread1.start(), thread2.start()  # Пестим място на слайда, не правете така
thread1.join(); thread2.join()    # Или така

print(f"Counter: {counter}")  # ???
```

Слайд (2) за конкурентност

```
def is_prime(n):
    if n < 2: return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0: return False
    return True

numbers = range(10**6, 10**6 + 500)
```

Колко **пъти** по-бърз е кодът отдясно?

```
start = time.time()
(is_prime(n) for n in numbers)
total = time.time() - start
print(total)
```

```
start = time.time()
with ThreadPoolExecutor(max_workers=4) as executor:
    executor.map(is_prime, numbers)
total = time.time() - start
print(total)
```

Хехе

```python
def is_prime(n):
    if n < 2: return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0: return False
    return True

numbers = range(10**6, 10**6 + 500)
```

Добре де, сега наистина - колко **пъти** по-бърз е кодът отдясно?

```python
start = time.time()

[is_prime(n) for n in numbers]

total = time.time() - start
print(total)
```

```python
start = time.time()

with ThreadPoolExecutor(max_workers=4) as executor:
    executor.map(is_prime, numbers)

total = time.time() - start
print(total)
```

Слайд (3) за конкурентност

И който е отговорил на предния въпрос, още един и получава точка:

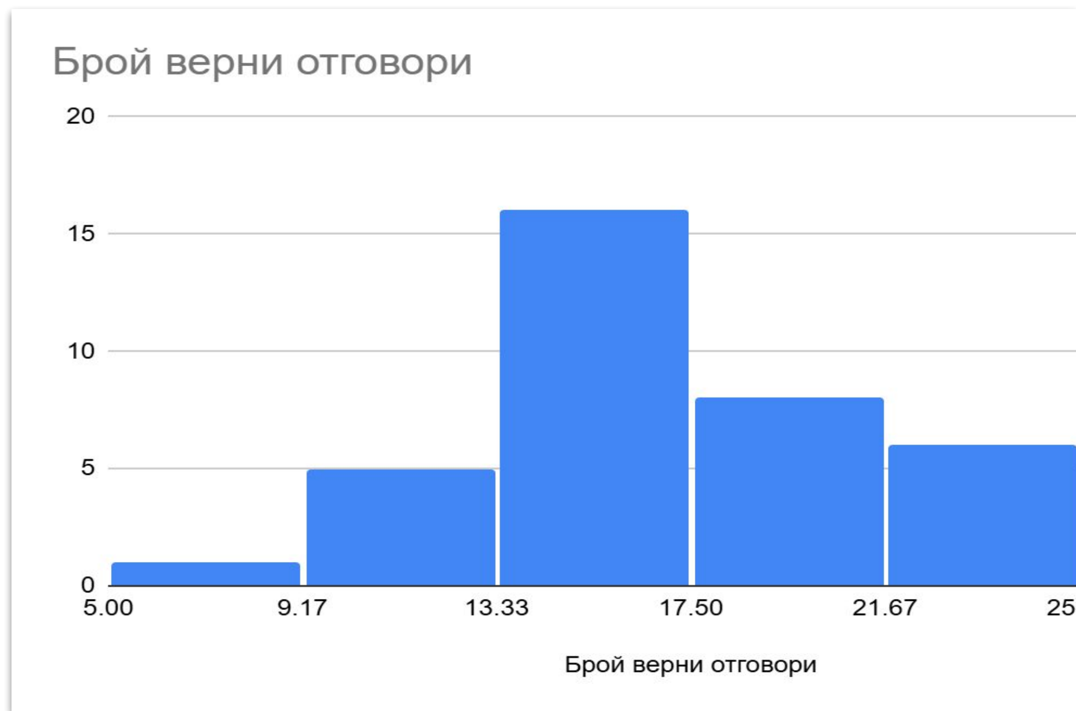
Каква е разликата между конкурентност и паралелизъм?

Конкурентност: Когато две изчисления нямат ясно дефинирана последователност на изпълнение.
Паралелизъм: Две изчисления, които реално се изпълняват едновременно.

Това го повтаряме, защото ще е важно и днес.

Слайд за предишното контролно

- Поискахте, получавате…
- Ама следващия път
- Защото днес има немалко материал, през който да минем
- Все пак, за clickbait-a:



Слайд за следващото контролно

- Дата за следващото контролно ще се избере на демократичен принцип:
  - Пускаме ви анкета
  - В анкетата ще има ПОНЕ 3 дати, всяка от които с ПОНЕ 3 опции за час
  - Гласувате за всеки час и дата, в които **можете** и ви е що-годе окей
  - А не за един, в който на вас ви е най-удобно
  - Идеята е да не се дублираме с други контролни и изпити, а не да не си изпуснете индийския сериал, така че бъдете колегиални
  - Накрая броим и за денят и часът, в който има най-много гласове и правим контролно тогава
- Сега въпрос - кога искате да ви пуснем анкетата, с други думи - кога ще знаете за останалите си контролни и изпити?

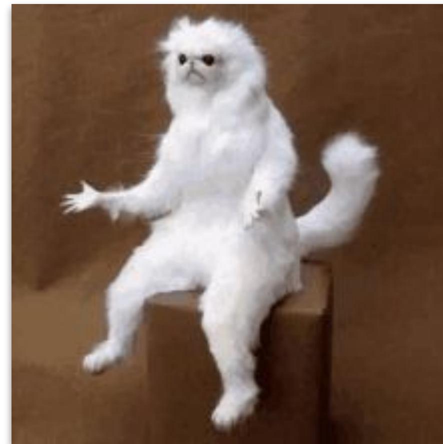
Слайдове за проектите

- Тъй като вече проектите са много на дневен ред, редно е да си поговорим малко повече по темата
- Ако останат въпроси - питайте

Кога?

- Защитите ще са на последната възможна дата, т.е. уикенда 4 *(събота)* и 5 *(неделя)* юли
- През деня
- Ще пуснем график в сайта, който позволява при mutual consent двама души да си разменят часовете
- Слотовете ще са от по 30-45 минути (предишни години сме ги правили по 30, но масово времето не ни е достигало, така че ще го премислим преди да пуснем графика), но се пригответе ако се наложи да изчакате

Как?



- Идвате с лаптопите си
- Говорим си за проектите ви
- Пишем ви точки **(0-60)**
- Смятаме ви финалните точки
- Пишем ви оценка
- Прибирате се
- Бангарангите
- Процесът е сравнително ясен, по-важни са изискванията

- Искаме от вас **задължително** да използвате GitHub за проектите си
- Не е нужно да ни пращате линкове предварително, не е нужно да са публични, но е задължително да са там
- И ако видим, че имате само 1 къмит - това не се брои
- Искаме от вас да **използвате** GitHub, а не просто да си качите проектите там като са готови

Оценяване

- Ще получите определен брой точки, в зависимост от някакви критерии, които почиват върху неща, които сме повтаряли през целия курс
- Накратко - функционалност, стил и тестове са трите големи области, които носят приблизително равен брой точки

Тестове

- Да, **тестове**, по-конкретно **unit тестове**
- Ще очакваме от вас да сте написали тестове, независимо дали проектът ви е игра, сайт, мобилно приложение или каквото още се сетите
- С добър дизайн - има как да напишете тестове за всяко едно нещо, без да тествате самия фреймуърк, който ползвате
- **insert пример**
- Това ще ви научи как да пишете тестваем код
- И не, нямаме претенции дали ползвате `unittest` или `pytest`

Среда

- Немалко неща казахме за виртуалните среди
- За питонските пакети
- За това как се менажират и каква е връзката между двете
- Имахме лайв демо, в което показахме как точно работи цялото нещо
- И да, там нямахме **реални** пакети, които ползвахме, но пък си ползвахме виртуалната среда и `git` както му е реда
- И ще гледаме за това във вашите проекти
- Може да не е огромен процент от оценката, но ще му обърнем внимание
- И не, не е нужно да избухвате с правене на пакети, изпълними файлове, мейкфайлове и прочие... Просто спазвайте нещата, които сме коментирали и показали на лекции
- Тези практики са универсални и ако решите да програмирате на Python - ще ни благодарите, че сме ви изтормозили

Слайд за каквото остана свързано с проектите

- За хората, които ще правят игра - ползвайте pygame-ce, а не pygame *(второто е на практика загинало)*
- Тази година не остана време да имаме лекция за дизайн, но истината е, че бихме я направили коренно различно, защото това е тема на цял курс, а не на една лекция
- Всичко по-малко от 2-3 лекции е просто отбиване на номера
- И тъй като предпочетохме да отделим време на други неща, тук можете да намерите лекцията от миналата година

Какво ви чака днес?

Преговор?

- user space VS kernel space
- preemption VS cooperation
  - yield

Преговор - нишки (операционни системи)

- Всяка нишка съответства на стек за изпълнение
- ... нарича се и системен стек
- ... защото обикновено е имплементиран от процесора
  - CALL/RET инструкции в x86
- Там живеят:
  - локалните променливи на функциите
  - аргументите им
  - адресът на извикващата функция (return address)
- Когато извикване (например IO) чака (*блокира*), нишката също чака
- За бъдат две блокиращи операции конкурентни, ни трябват две нишки
- Нишките ни дават preemptive multitasking
- Имплементира се от OS-a...
- ... чрез процесорни прекъсвания за време

Преговор - прекъсвания (компютърни архитектури)

- Комуникацията с хардуера става чрез прекъсвания (interrupts)
- Те са не-блокиращи (асинхронни)
- Има глобална таблица с код за обработка на прекъсвания
- Всичко това живее в ядрото (kernel space)
- … и не е директно достъпно за потребителски програми (user space)
- Част достигат до user space чрез системни извиквания (syscalls)
  - `INT 21h` в x86/DOS
  - `ENTER`/`LEAVE` в по-съвременен x86

blocking VS async

- Нишките са блокиращи
- Прекъсвания са асинхронни
- Операционната система скрива асинхронноста от нас
- ... чрез комбинация от busy loops + инструкции за изчакване
    - `HLT` (halt) в x86
    - `WFI`/`WFE` (wait for interrupt/wait for event) в ARM
    - ... и т.н.

Нишки - предимства

- Простота!
  - Може да не вярвате че нишките са прости...
  - ... алтернативите ще ви накарат да размислите

Нишки - недостатъци

- Стартирането им не е безплатно откъм време
- Нужно е да резервираме [последователна] памет за стек
  - резервираме != алокираме (първото не е толкова зле)
- Горните две могат да се решат с thread pooling
  - Преизползване на вече създадени нишки
- Време за превключване между нишки
  - Трябва да се мине през kernel-a
  - Обвързано е с context switch
  - Обикновено – 2-5 μs
  - Може да достигне до 200μs (милиони тактове на процесора)
  - Meltdown?!
  - Ако нишките са хиляди, цената се трупа
  - Възможно ли е да обработим десетки хиляди HTTP request-и за секунда?

Нишки - решения

- Исторически има различни решения
- Като например...

Зелени нишки

- Всяка зелена нишка съответства на стек за изпълнение
- Имплементират се като част от библиотеката на езика
- Редуват се в използването на [по-малко количество] системни нишки
  - m:n – threads:green-threads (m < n)
  - Scheduling-a се имплементира в user space от runtime-a на езка
- Обикновено са preemptive
  - ... но блокиращите операции могат служат за предаване на изпълнението (yield)
- Облекчават проблема с блокирането
- Имплементират се чрез fibers (или подобен механизъм)

fibers

- Всяки fiber съответства на стек за изпълнение
- Понякога са част от езика
  - … понякога ги има и в OS-a (win32)
  - … могат и да се имплементират в user space
- Можем да ги “замразим”
- Няма scheduling – потребителят сам предава изпълнението

Python

- В python няма нито green threads, нито fibers
- Използват се системните нишки
- Не е гаранция, че в бъдеще няма да се появят
  - За справка Project Loom в Java 21
  - … 27 години след създаването на езика

<offtopic>coroutine</offtopic>

- Като fiber
- Без собствен системен стек за изпълнение
- Не могат да се замразят в произволен момент
- Целта е да запазим състоянието на изпълнението
- Концептуално приличат на генератор в Питон...

```
def coroutine():
    for task in get_tasks():
        yield process_tasks(task)
```

- `yield`-а служи за... yield
- `yield` е единственото място, на които изпълнението може да спре
- ... в по-стари версии генераторите дори се използват като корутини!

Някога много, много отдавна

- ... това беше валиден код:

```
import asyncio
import random


@asyncio.coroutine
def slow_operation():
    sleep = random.choice(range(1, 6))
    print('Starting slow operation')
    yield from asyncio.sleep(sleep)
    print('Finished slow operation')
    return 'A return value from the slow_operation coroutine took %s seconds' % sleep


def got_result(future):
    print('Requesting result')
    print(future.result())
    print('Got result')
```

Зелени нишки + блокиране

- Зелените нишки имат огромен смисъл, когато се изчакват една друга
- Колко зелени нишки вървят, ако системните са блокирани?
- Николко!

Зелени нишки + асинхронност

- Ядрото използва асинхронност
- А ако имахме асинхронност в user space?
- Има!
  - Linux - `epoll`
  - Linux - `io_uring`
  - BSD/OSX - `kqueue`
  - Windows - IOCP (I/O Completion Ports)
- Понякога няма:
  - Dos
- Понякога е минимална:
  - Unix (System V)

Зелени нишки + асинхронност - зелени нишки

- Нужни ли са ни въобще зелени нишки?
  - Не!
  - … не пречи да ги имаме
  - … но Питон ги няма
- Достатъчни са ни…
  - една системна нишка
  - асинхронност
  - някаква форма на корутини (за да пазим докъде е стигнало изпълнението)

User space асинхронност

- Недостатъци на async kernel API-тата
  - от много ниско ниво
  - не са унифицирани между операционните системи
  - понякога не са добри (`epoll` VS `io_uring` под Linux)
- Очакваме езикът да предостави API
  - от високо ниво
  - унифицирано между операционни системи
- ... а именно...

asyncio

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number+1):
        print(f"Task {name}: Compute factorial({i})...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")

async def main():
    tasks = [
        asyncio.create_task(factorial("A", 2)),
        asyncio.create_task(factorial("B", 3)),
        asyncio.create_task(factorial("C", 4)),
        asyncio.create_task(factorial("D", 10)),
    ]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

Магия

- Всяка функция с async има
- Всичко се случва в една нишка
- await позволява няколко sleep-а да чакат “едновременно”
- Същото важи за IO операции

```
>>> type(main)
<class 'function'>
>>> type(main())
<python-input-8>:1: RuntimeWarning: coroutine 'main' was never awaited
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
<class 'coroutine'>
```

asyncio + нишки

- Можем да направим блокиращ код асинхронен

```
import asyncio
import time

def blocking():
    time.sleep(2)
    return 42

async def main():
    loop = asyncio.get_running_loop()
    result = await loop.run_in_executor(None, blocking)
    print(result)

asyncio.run(main())
```

Meta

- `__aiter__`
- `__anext__` (`AsyncStopIteration`)

async for

```
class AsyncRange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __aiter__(self):
        return self

    async def __anext__(self):
        if self.i >= self.n:
            raise StopAsyncIteration
        await asyncio.sleep(1)   # асинхронна пауза
        value = self.i
        self.i += 1
        return value

async def main():
    async for x in AsyncRange(3):
        print("Got:", x)

asyncio.run(main())
```

Meta (2)

- `__aenter__`
- `__aexit__`

async with

```
class AsyncResource:
    async def __aenter__(self):
        print("Opening resource...")
        await asyncio.sleep(1)
        return "resource"

    async def __aexit__(self, exc_type, exc, tb):
        print("Closing resource...")
        await asyncio.sleep(1)

async def main():
    async with AsyncResource() as r:
        print("Using", r)

asyncio.run(main())
```

Blocking vs async

Нека сравним два прости HTTP сървъра:

- блокиращ
- асинхронен

Блокиращ сървър

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class Handler(BaseHTTPRequestHandler):
    def log_message(self, format, *args):
        pass

    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/plain")
        self.end_headers()
        self.wfile.write(b"Hello World")

HTTPServer(("0.0.0.0", 8091), Handler).serve_forever()
```

Асинхронен сървър

```python
import asyncio

async def handle(reader, writer):
    response = b"HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\nHello World"
    writer.write(response)
    await writer.drain()
    writer.close()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(handle, "0.0.0.0", 8091)
    async with server:
        await server.serve_forever()

asyncio.run(main())
```

bash benchmark

```
strace -c -f -o strace.out python server.py &
STRACE_PID=$!
sleep 1
SERVER_PID=$(pgrep -P $STRACE_PID python)

for i in {1..1000}; do
    curl -s http://localhost:8091/ > /dev/null 2>&1 &
    CURL_PIDS="$CURL_PIDS $!"
done

echo "Wait only for curl jobs"
for pid in $CURL_PIDS; do
    wait $pid
done

kill $SERVER_PID
wait $STRACE_PID

echo "=== Syscall summary ==="
cat strace.out
```

Време за 1000 заявки

- Блокиращо
```
real    0m3.336s
user    0m4.121s
sys     0m5.638s
```
- Асинхронно
```
real    0m1.761s
user    0m4.236s
sys     0m5.777s
```

Извод

- При една нишка, асинхронният код е по-бърз

Blocking vs async vs threaded

Нека добавим още един сървър към сравнението:

- Блокиращ, но с thread pool

Многонишков сървър

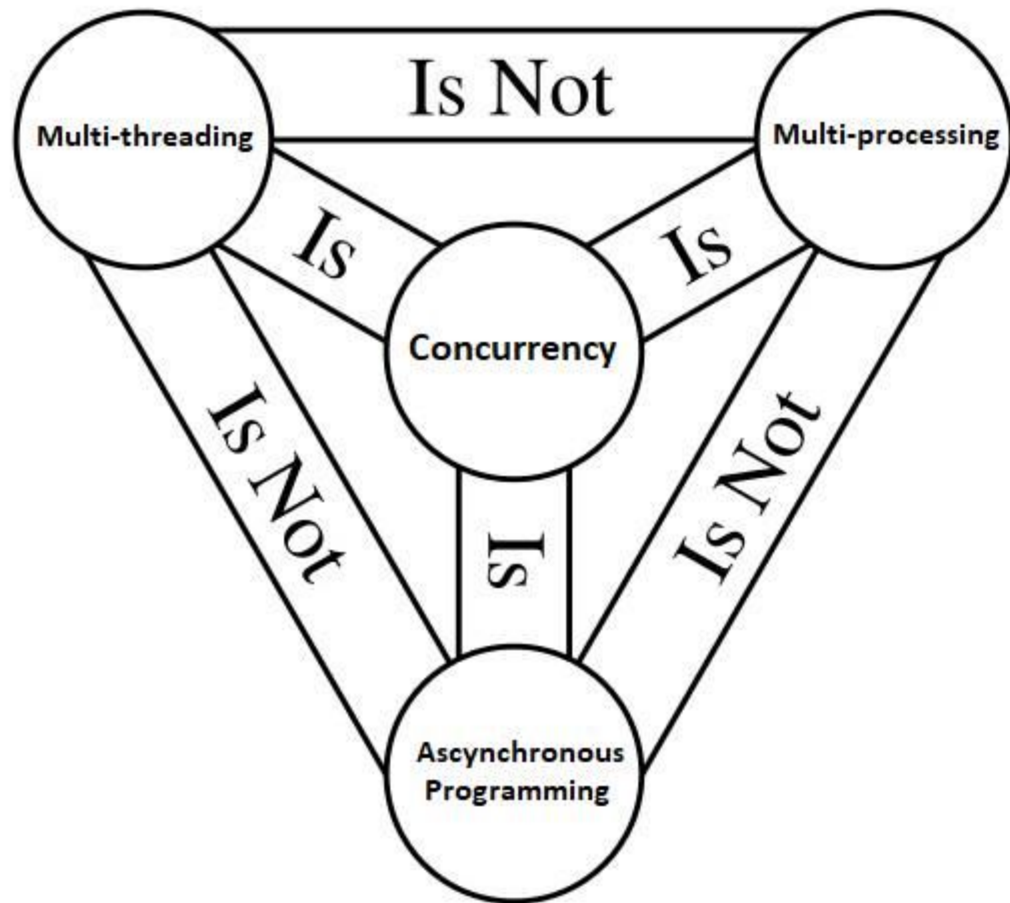
```python
import socket
import threading
from http.server import BaseHTTPRequestHandler, HTTPServer
from concurrent.futures import ThreadPoolExecutor

class Handler(BaseHTTPRequestHandler):
    def log_message(self, format, *args):
        pass  # silence all logs

    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-Type", "text/plain")
        self.end_headers()
        self.wfile.write(b"Hello World")

class ThreadPoolHTTPServer(HTTPServer):
    def __init__(self, server_address, handler_class, num_workers):
        super().__init__(server_address, handler_class)
        self.executor = ThreadPoolExecutor(max_workers=num_workers)

    def process_request(self, request, client_address):
        self.executor.submit(self._handle_request_noblock, request, client_address)

if __name__ == "__main__":
    server = ThreadPoolHTTPServer(("0.0.0.0", 8080), Handler, num_workers=8)
    server.serve_forever()
```

Време за 1000 заявки

- Блокиращо
```
real    0m3.336s
user    0m4.121s
sys     0m5.638s
```
- Асинхронно
```
real    0m1.761s
user    0m4.236s
sys     0m5.777s
```
- Блокиращо + многонишково
```
real    0m5.681s
user    0m4.786s
sys     0m7.586s
```

Изводи

- Многонишковият може да бъде по-бавен от еднонишковия
  - Как и защо???
  - Цена за поддържане на множество нишки
  - GIL
- Щеше да бъде по-бърз, ако:
  - Имаше нещо повече от `write(b"Hello World")`
  - Например нещо синтетично като `sleep`
  - … или нещо по-реалистично като достъп до база данни


Is Not
Multi-threading
Multi-processing
Is
Is
Concurrency
Is Not
Is
Is Not
Ascynchronous
Programming

Въпроси?