СЛЁРМ Обучение для инженеров и технических лидеров

Дата-инженер

Параллельность и асинхронность в Python

Николай Марков

Цели урока. Что вы узнаете:

1. Что такое потоки и процессы
2. Что такое GIL и как все-таки работать с потоками в Python
3. Как работает асинхронность и чем она отличается от параллельности
4. Как asyncio в Python позволяет нам писать эффективный код

Николай Марков

О спикере

- Занимаюсь профессиональной разработкой и проектированием уже больше 11 лет
- Начинал программистом на Python, сейчас Principal Architect в компании Aligned Research Group, а также ментор в компании DryLabs
- В разное время работал с сетями, протоколами и различными облаками (AWS, GCP, Azure, OpenStack)
- Писал проекты на Python, а также Golang, C/C++, Scala и Rust
- Сейчас выстраиваю аналитические архитектуры и Data Governance в разных компаниях

Что такое потоки и процессы?

Как работает ОС

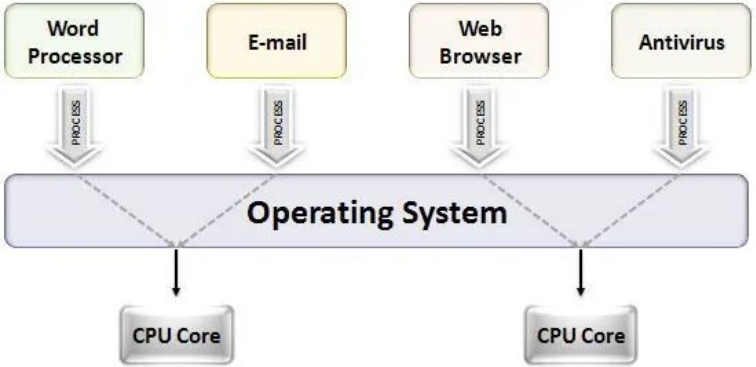
В системе запущено много приложений, и системный планировщик должен каждому выделить время и ресурсы

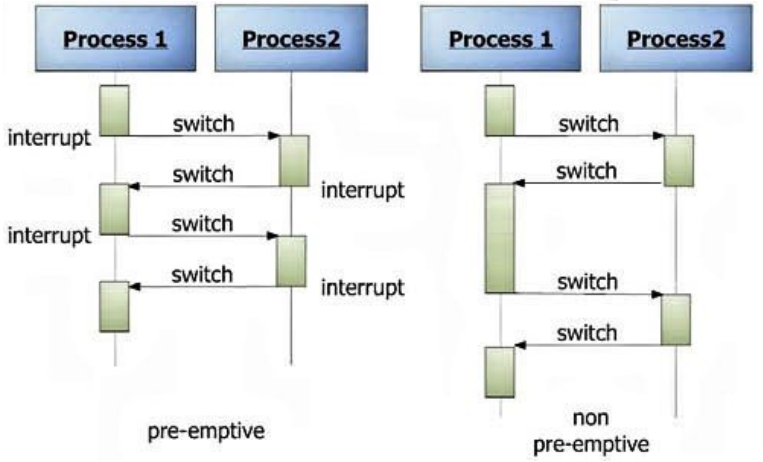
Каждое приложение состоит из процесса или даже группы процессов, внутри каждого из которых может быть несколько потоков

ОС выделяет область памяти под каждый новый процесс, при этом все потоки этого процесса имеют к ней доступ одновременно

В английском языке есть два термина **stream** и **thread**, которые оба переводятся на русский, как **«поток»**. Мы здесь говорим о потоках именно в смысле «thread»

Многозадачность





Вытесняющая многозадачность
(используется в большинстве современных ОС)

Кооперативная многозадачность
(используется в старых и специализированных ОС)

Потоки непредсказуемы

Системный планировщик отдает процессорное время потокам внутри процессов, **переключая между ними контекст**

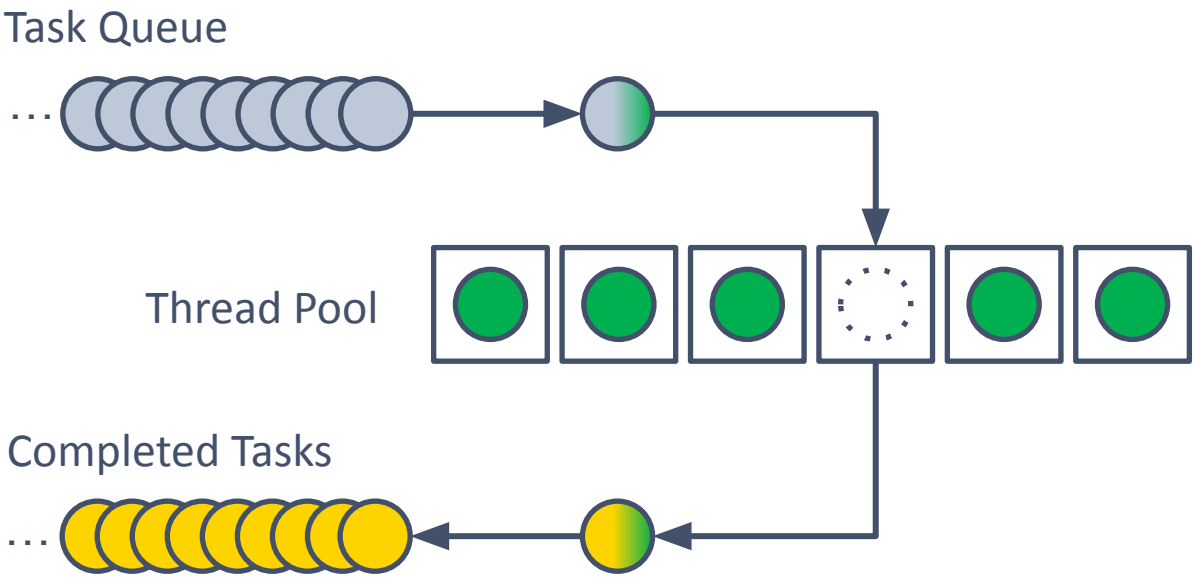
Практически **невозможно заранее предсказать**, какой процесс/поток получит ресурсы в конкретный момент

Потоки работают **«параллельно»**, в идеале **используя несколько ядер** процессора

Потоки надо **синхронизировать** согласно задачам, чтобы не было **проблем с одновременным доступом**

Параллельное программирование — довольно сложная область, которую разные языки пытаются упростить, изобретая разные модели и парадигмы

Пул потоков



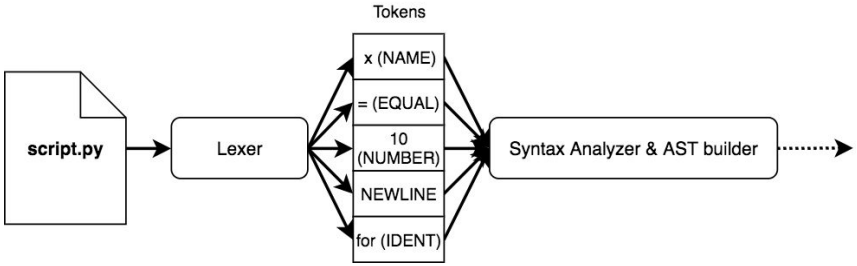
Что такое GIL?

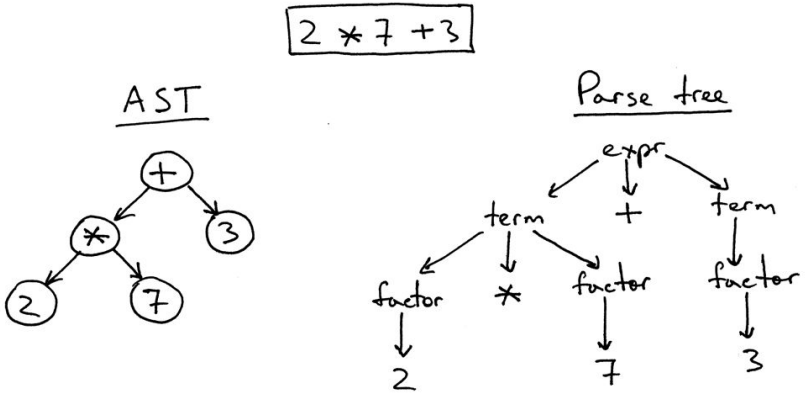
Global Interpreter Lock

- GIL — это **глобальный мьютекс** (механизм синхронизации) в интерпретаторе Python
- GIL запрещает выполнять байткод Python **больше, чем одному потоку одновременно**
- Но это касается **ТОЛЬКО** байткода Python и **не распространяется на I/O операции**
- Потоки Python (в отличие от потоков, скажем, в некоторых версиях Ruby) — это **полноценные потоки ОС**

  Визуализация от Дэвида Бизли

Почти в любом языке программирования





Ассемблер или байткод

```
  0 LOAD_CONST               1 (2)
  3 LOAD_CONST               1 (2)
  6 BINARY_MULTIPLY
  7 PRINT_ITEM
  8 PRINT_NEWLINE
  9 LOAD_CONST               0 (None)
 12 RETURN_VALUE
```

Примеры параллельного кода

Примеры многопоточного кода

В Python есть два взаимозаменяемых механизма для работы
с потоками — модули threading и concurrent.futures

https://pastebin.com/iwA5uyVV

https://pastebin.com/VEhbv3jQ

А можно просто взять процессы?

```
1.  from multiprocessing import Process
2.  from multiprocessing import Queue
3.  concurrent.futures.ProcessPoolExecutor
```

В случае использования процессов возникают большие затраты на пересылку данных и сериализацию.
Но над этим уже ведется работа!

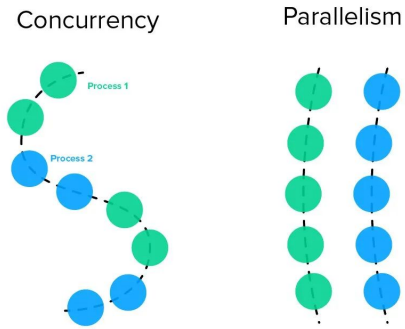
Асинхронность?
Параллельность?
А может быть, конкурентность?

Это не конкурирующие понятия (поняли, да?)

Параллельность — это когда два фрагмента кода выполняются **физически одновременно**, например, на разных ядрах CPU

Асинхронность — это когда код выполняется **НЕ последовательно**, как написан в файле, а в виде реакций (**callback**'ов) на события.
Это может происходить параллельно, а может нет.

Конкурентность — это общее название ситуации, когда несколько задач выполняются **НЕ в строго определенном порядке**, а иногда
и с перехлестом по времени

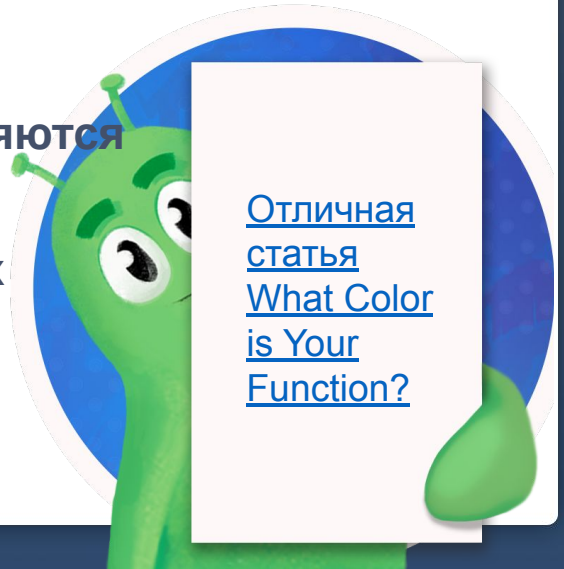


Корутины и event loop

- Асинхронность может быть реализована с помощью **параллельности** или **ручного переключения** контекста в коде, сохраняя последнее состояние.
  Кто сказал **yield**?
- «**Кооперативной многозадачность**» - фрагменты кода (корутины или сопрограммы) **самостоятельно определяют**, **когда передавать управление друг другу** и не зависят от внешнего системного планировщика.
- Нужна дополнительная сущность, чтобы **связывать фрагменты кода с событиями**, которая называется **event loop.**
- Недостаток — долгоиграющая процедура **НЕ под контролем** event loop вешает ВСЕ процессы приложения.

Событийно ориентированное программирование

- Пока корутина **ждет внешнее событие**, контекст переключается на другую
- Сообщение о переключении контекста может содержать в себе **данные** для другой корутины
- В современной реализации асинхронности в Python **обычные и асинхронные функции не являются взаимозаменяемыми**
- Существуют альтернативные реализации для старых версий — **Gevent**, **Eventlet** и **Tornado** и др



Отличная статья What Color is Your Function?

Классические генераторы

```
1.  import random
2.
3.  def multiply_gen(lst):
4.      multiplier = 2
5.      for i in lst:
6.          multiplier = (yield multiplier) or multiplier
7.          print(f"Multiplier is now {multiplier}")
8.
9.  data = [random.randint(-10, 10) for _ in range(10)]
10. mult_data = multiply_gen(data)
11. for i, entry in enumerate(zip(data, mult_data)):
12.     print(f"{i}: {entry} {entry[0] * entry[1]}")
13.     if i == 4:
14.         mult_data.send(3)
15.         print(f"{i}: {entry} {entry[0] * entry[1]}")
```

У все того же Дэвида Бизли есть отличные слайды как про генераторы, так и про корутины в классическом понимании

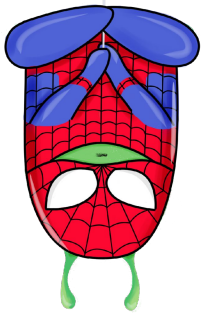
Asyncio

Asyncio

```
1.  import asyncio
2.
3.  asyncio.Queue()  # асинхронная очередь
4.  asyncio.sleep(10)  # асинхронный "сон"
5.  # асинхронный subprocess
6.  asyncio.create_subprocess_exec()
7.  asyncio.Lock()  # асинхронный мьютекс
8.  # ручное добавление корутины в event loop
9.  asyncio.ensure_future()
10. # дождаться окончания работы списка корутин
11. asyncio.gather()
```

Ключевые слова async/await

```
1.  import asyncio
2.
3.  async def hello(name):
4.      return "Hello, {}!".format(name)
5.
6.  async def call_vasya():
7.      greeting = await hello("Vasya")
8.      return greeting
9.
10. loop = asyncio.get_event_loop()
11. print(loop.run_until_complete(call_vasya()))
```



Когда мы можем подождать?

Чтение и **запись байтов** в любое устройство осуществляется через **системный вызов**

Каждый раз, когда ответ на наш запрос зависит от **внешних (по отношению к нашему коду) обстоятельств** — мы можем смело явно **переключить контекст** на другие задачи

Сама возможность переключать контекст вручную **без оглядки на механизм потоков** позволяет утилизировать ресурсы системы **намного эффективнее**, чем оно работало бы через системный планировщик

Важный подводный камень состоит в том, что если поддержка переключения контекста **не реализована** в библиотеке— есть шанс **подвесить все приложение.**
Но на случай, когда все плохо, в Python есть специальная обертка

Справляемся с проблемами

Как запустить дочерний процесс, при этом оставив возможность посылать ему данные на стандартный ввод, читая его стандартный вывод?

https://github.com/aio-libs

Работа с диском

Асинхронная работа с диском

До недавнего времени большинство операционных систем в **не умело работать с файлами на диске асинхронно.**

В большинстве языков программирования, включая Python, поддержка асинхронности реализуется с помощью потоков.

В ядре Linux (начиная с 5.4) есть механизм io_uring, он позволяет решать проблему. Ждем удобных интерфейсов!

Итоги. О чем поговорили:

1. Что такое потоки и процессы
2. Что такое GIL, мешает он нам или помогает
3. Как писать многопоточный и многопроцессный код на Python
4. В чем разница между асинхронностью, параллельностью и конкурентностью
5. Зачем нужен модуль asyncio и парадигма async/await

Спасибо
за внимание!