asyncio

Злата Обуховская,
Team lead Nvidia, http://bigdatateam.org/
https://www.facebook.com/bigdatateam/

27.01.2020, Moscow, Russia

BIGDATA TEAM

Содержание

I. Что такое asyncio
II. Асинхронность, конкурентность, параллелизм
III. Как появился asyncio
IV. Как работает asyncio
V. Асинхронные библиотеки в питоне

I. Что такое asyncio?

asyncio — это библиотека для асинхронного ввода-вывода

Пример синхронной загрузки картинок:

https://github.com/haruhara/bigdata_course_asyncio/blob/master/01_scraper_sync.py

Пример асинхронной загрузки картинок:

https://github.com/haruhara/bigdata_course_asyncio/blob/master/02_scraper_async.py

asyncio помогает ускорить задачи, где идет работа с несколькими сетевыми соединениями

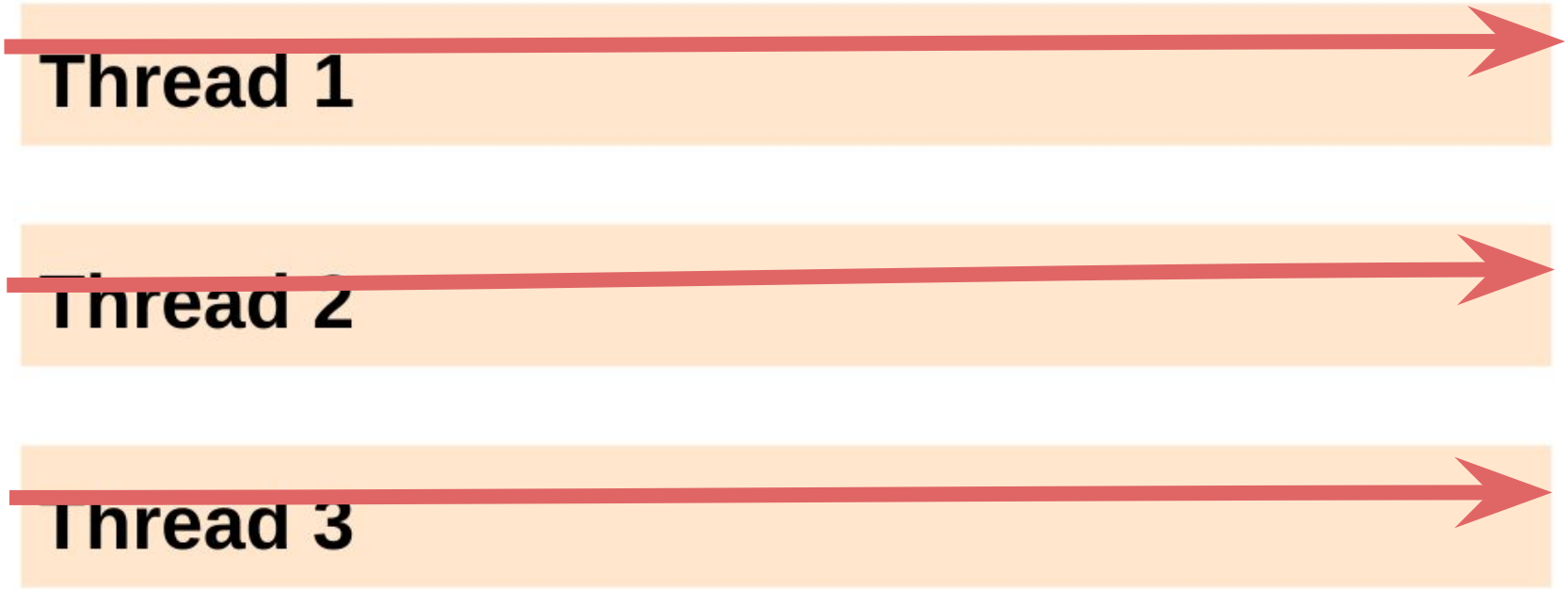
Попытка «распараллелить» вычислительную задачу при помощи asyncio:

https://github.com/haruhara/bigdata_course_asyncio/blob/master/03_fibonacci.py

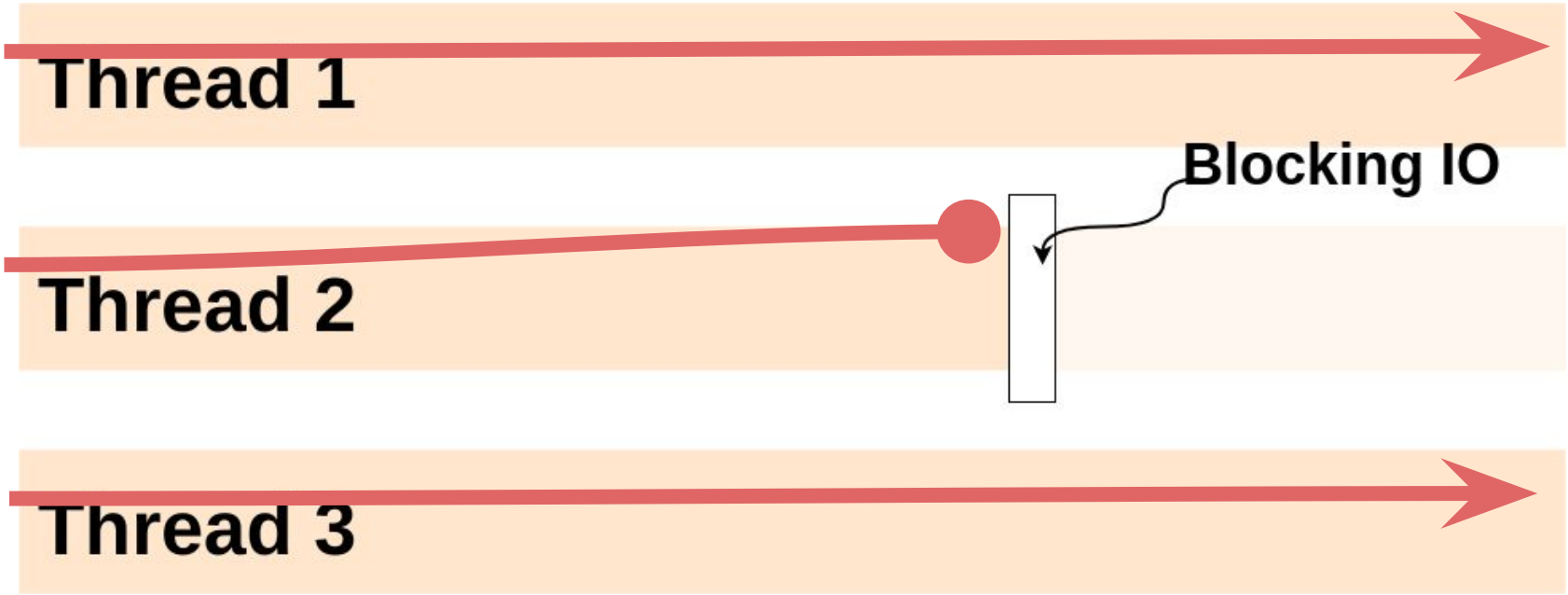
asyncio не помогает ускорить вычислительные задачи

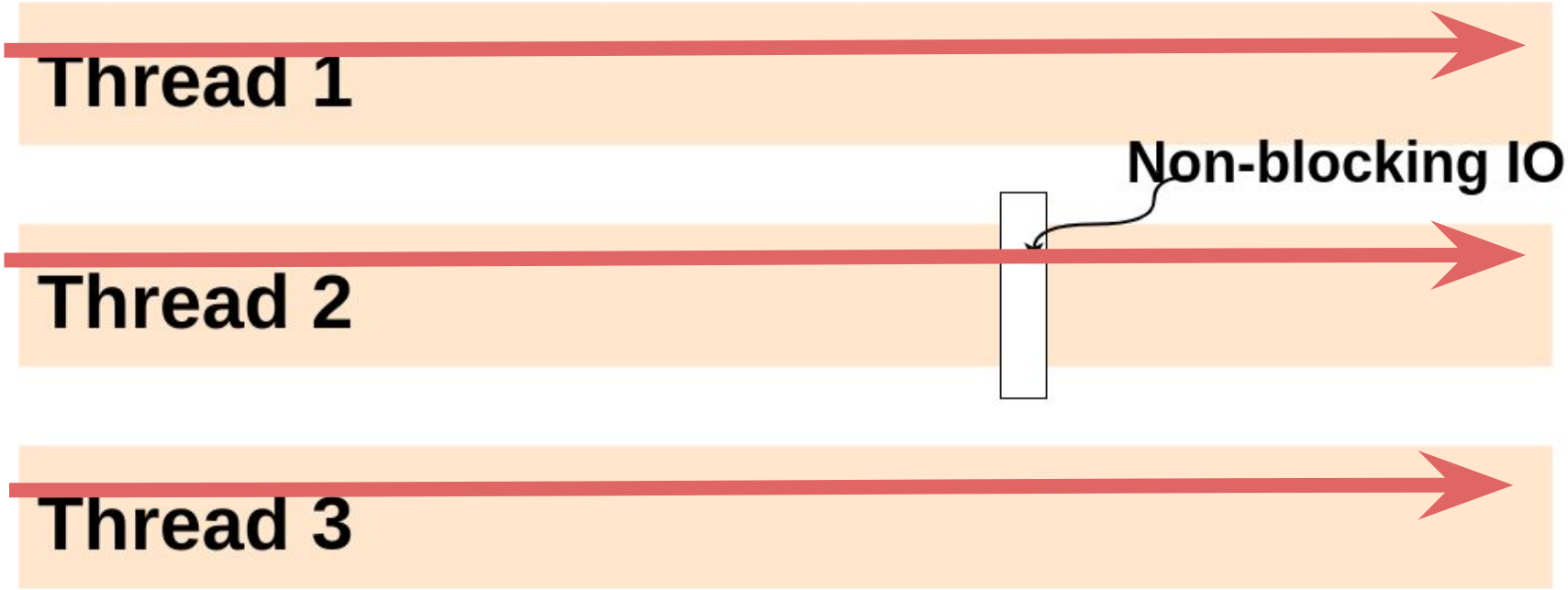
II. Асинхронность, конкурентность, параллелизм

Параллелизм — выполнение частей кода одновременно


Thread 1
Thread 2
Thread 3

Асинхронность означает неблокирующий ввод-вывод


Thread 1
Blocking IO
Thread 2
Thread 3


Thread 1
Non-blocking IO
Thread 2
Thread 3

Конкурентность — это не параллелизм

Конкурентность:

- разбиение кода на независимо выполняемые части
- свойство языка

Параллелизм:

- исполнение частей кода одновременно
- свойство среды выполнения

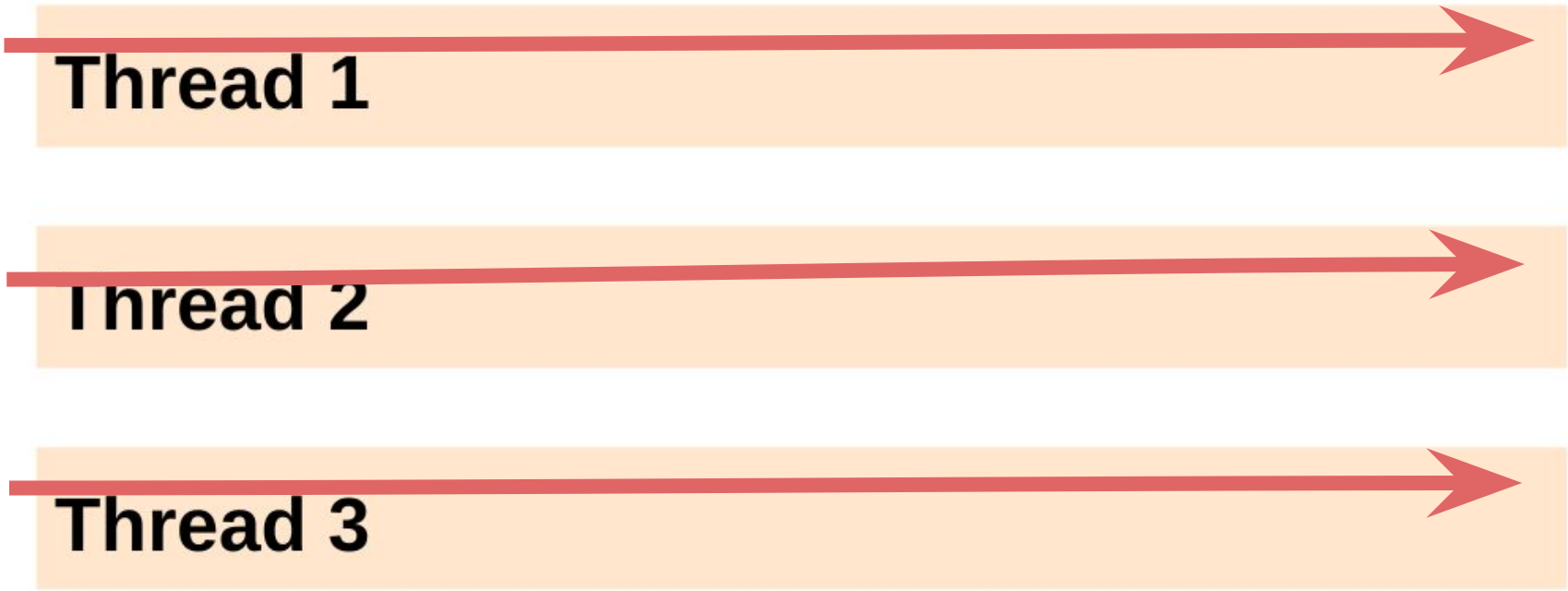
III. Как появился asyncio?

Типы конкурентности в питоне 2:

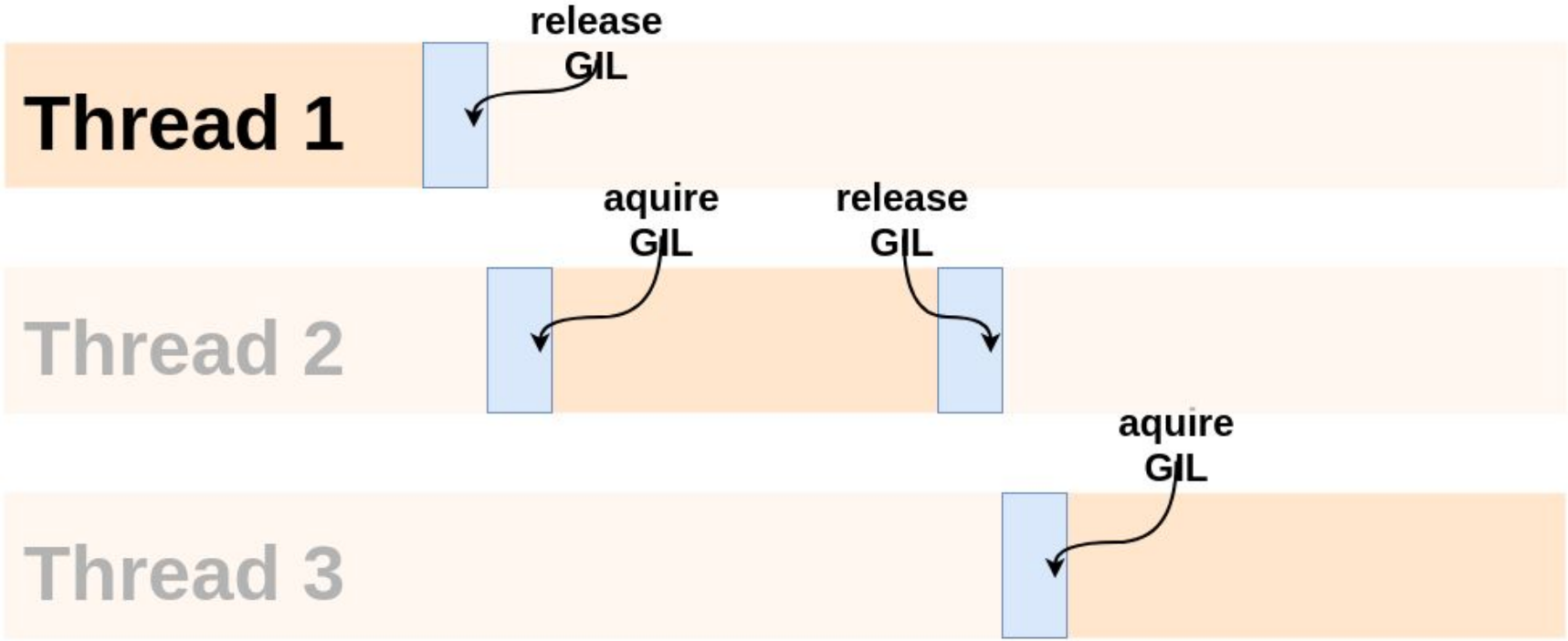
- Стандартные библиотеки multiprocessing, threading
- Callback-библиотеки (Twisted, Tornado)
- Green-threads, они же coroutines (Gevent, evenlet)

Проблема threading —

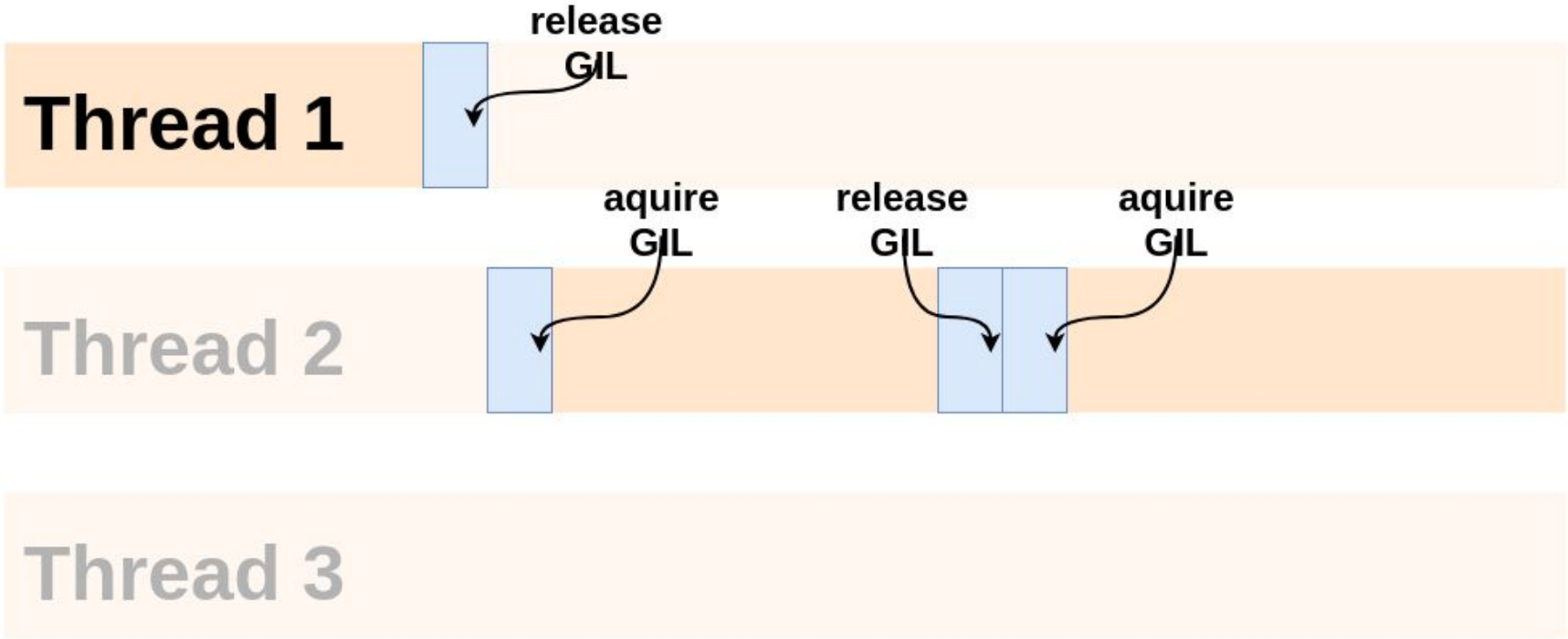
Global Interpreter Lock (GIL)


Thread 1
Thread 2
Thread 3

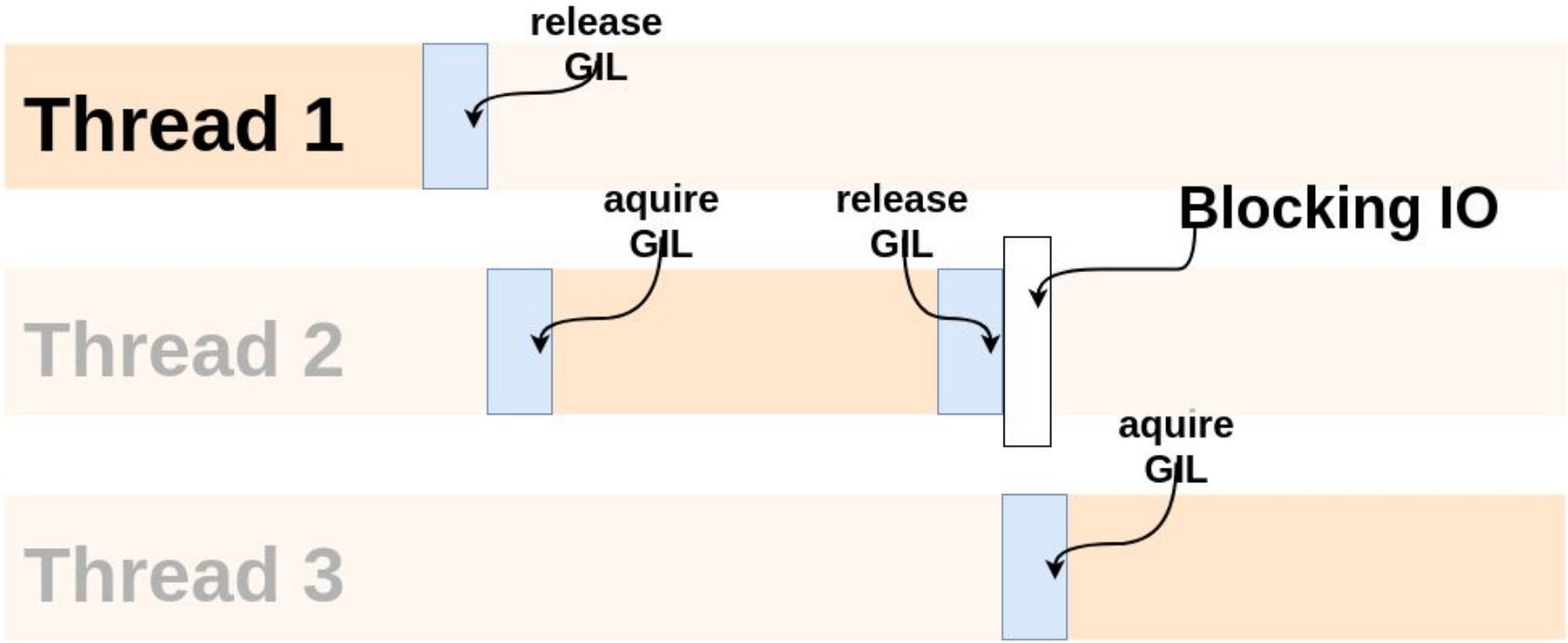
В питоне нет параллельного выполнения потоков

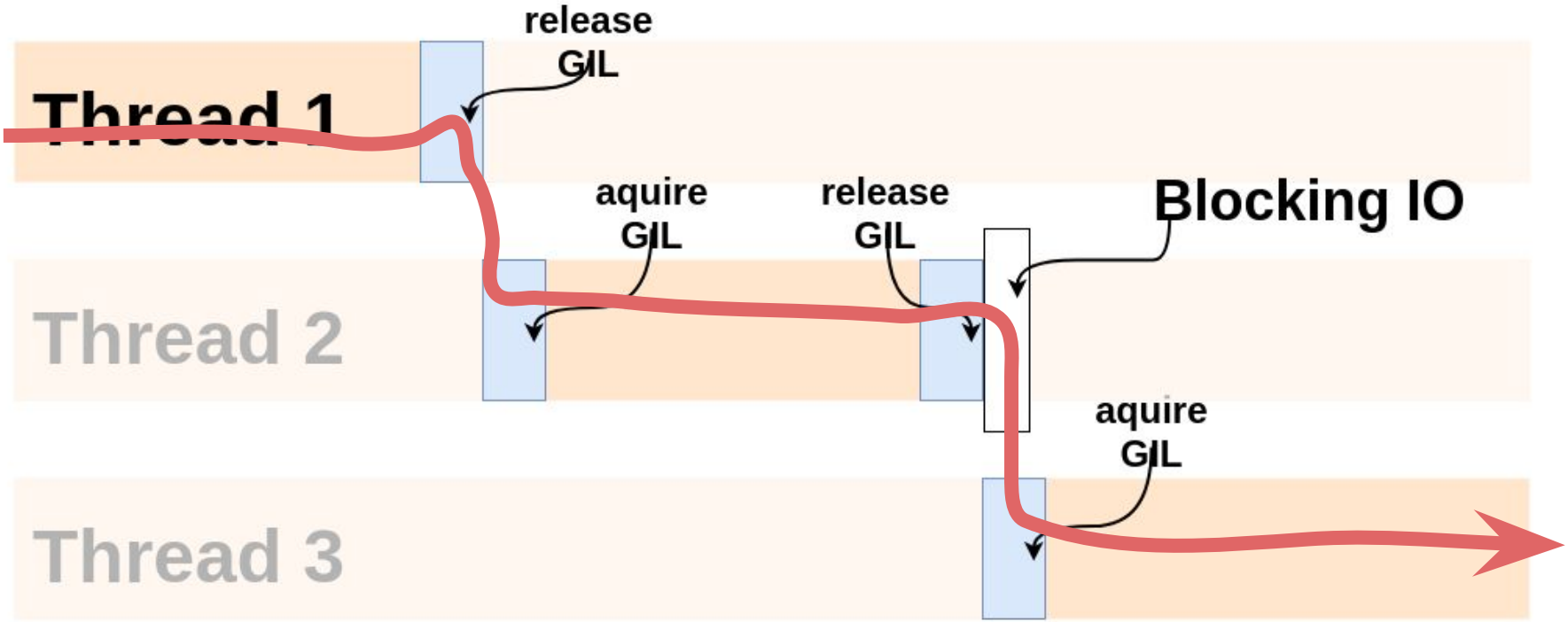

release
GIL
Thread 1
aquire
GIL
release
GIL
Thread 2
aquire
GIL
Thread 3

Иногда GIL препятствует равноправному выполнению потоков


release
GIL
Thread 1
aquire
GIL
release
GIL
aquire
GIL
Thread 2
Thread 3

Модуль threading используют, когда без блокирующего IO никак не обойтись


release
GIL
Thread 1
aquire
GIL
release
GIL
Blocking IO
Thread 2
aquire
GIL
Thread 3


release
GIL
Thread 1
aquire
GIL
release
GIL
Blocking IO
Thread 2
aquire
GIL
Thread 3

BIGDATA TEAM

Пример: скрипт для сбора данных

```
import urllib2

urls = ['http://www.python.org', 'https://docs.python.org/2/']

def collect(url):
    data = urllib2.urlopen(url).read()
    return data

big_data_dump = [collect(url) for url in urls]
```

BIGDATA TEAM

Callbacks (на примере Tornado)

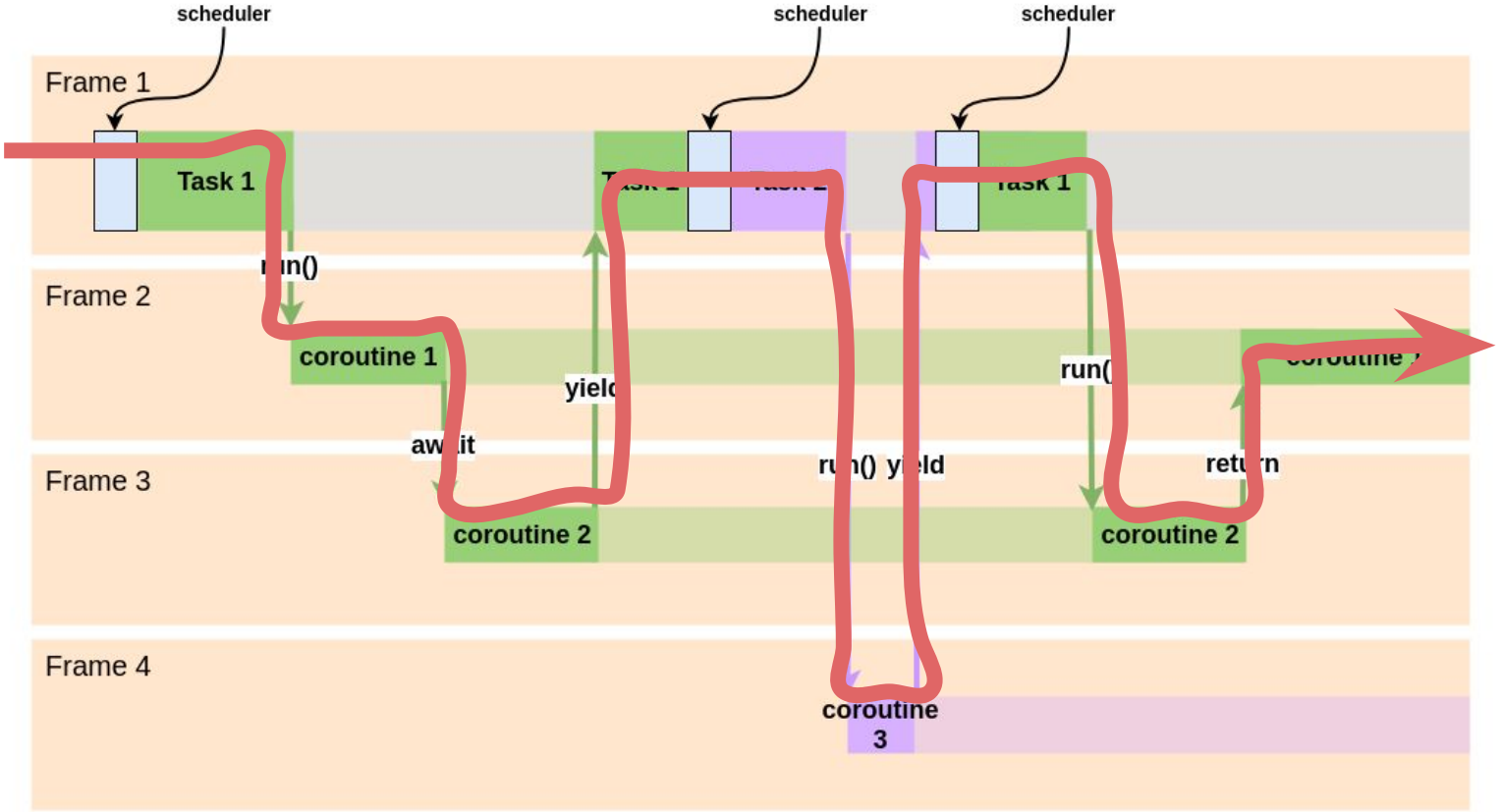
```
import tornado.ioloop
from tornado.httpclient import AsyncHTTPClient

urls = ['http://www.python.org', 'https://docs.python.org/2/']

def collector(response):
    data = response.body
    print data

http_client = AsyncHTTPClient()

for url in urls:
    http_client.fetch(url, collector)

tornado.ioloop.IOLoop.instance().start()
```

Event loop — бесконечный цикл, в котором вызываются обработчики событий (callbacks). Например, событий о готовности сетевых соединений

Coroutines (на примере Gevent)

```
import gevent
import gevent.monkey

gevent.monkey.patch_all()
urls = ['http://www.python.org', 'https://docs.python.org/2/']

def collect(url):
    data = urllib2.urlopen(url).read()
    return data

jobs = [gevent.spawn(collect, url) for url in urls]

big_data_dump = [result.value for result in gevent.joinall(jobs)]
```

Coroutine — функция, которая может передать поток выполнения в другую функцию, а потом начать выполнение с того же места


scheduler
scheduler
scheduler
Frame 1
Task 1
Frame 2
coroutine 1
Frame 3
coroutine 2
Frame 4
coroutine 3
run()
yield
await
run()
return
coroutine 2

В питоне 3.4 (2015 год) появились:

- Event loop вместе с модулем asyncio
- На уровне интерпретатора — coroutines
- Вместо callbacks — Tasks в asyncio

Пример асинхронного сервера для загрузки картинок:

https://github.com/haruhara/bigdata_course_asyncio/blob/master/04_scraper_server.py

IV. Как работает asyncio?

Корутины — на уровне интерпретатора
Event loop — на уровне библиотеки asyncio

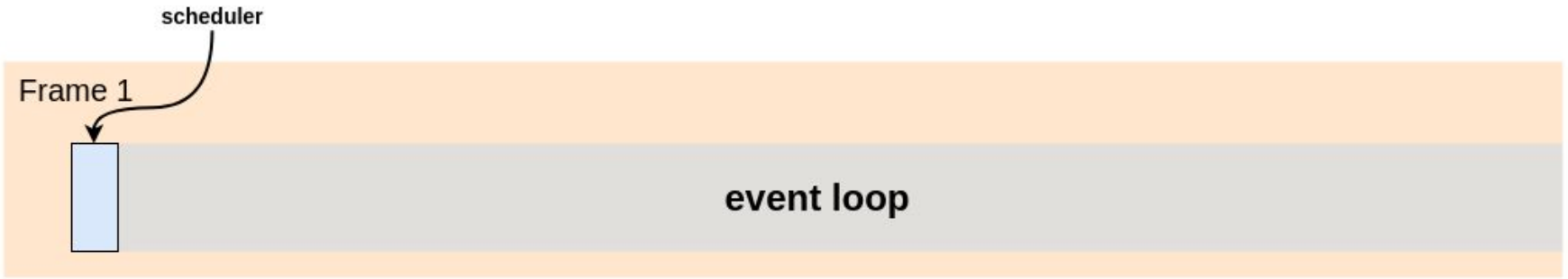
Frame 1

event loop

Event loop в asyncio — бесконечный цикл, где происходит выполнение объектов типа Task

Task становится готов к выполнению, когда:

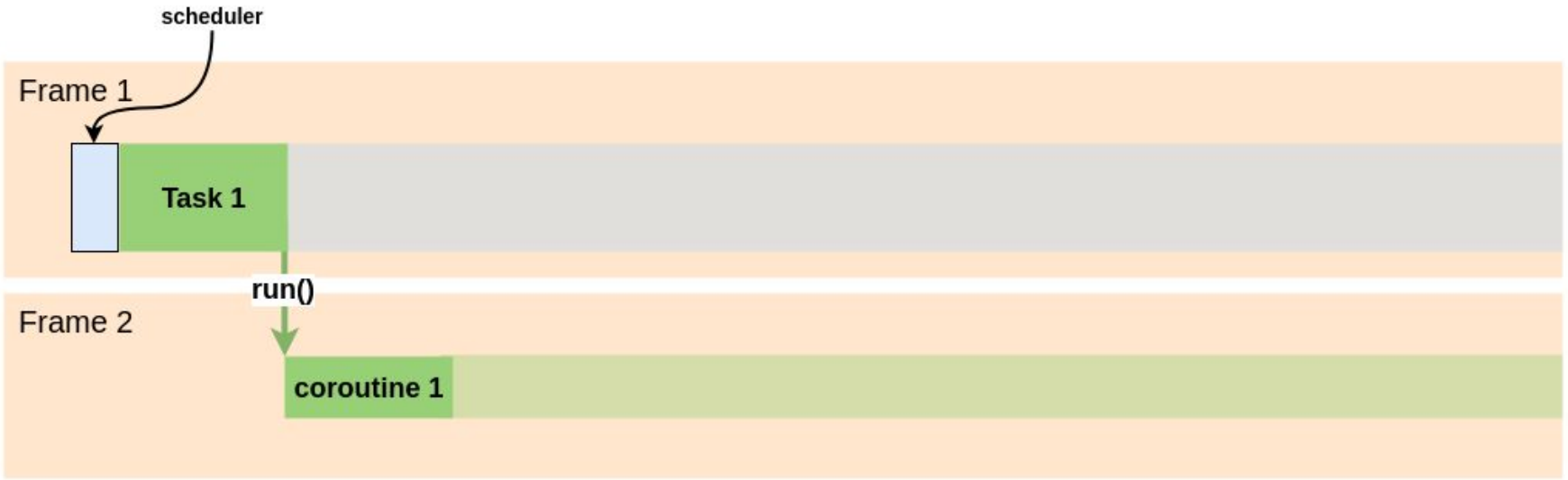
- готово соединение
- сработал таймер


scheduler
Frame 1
event loop

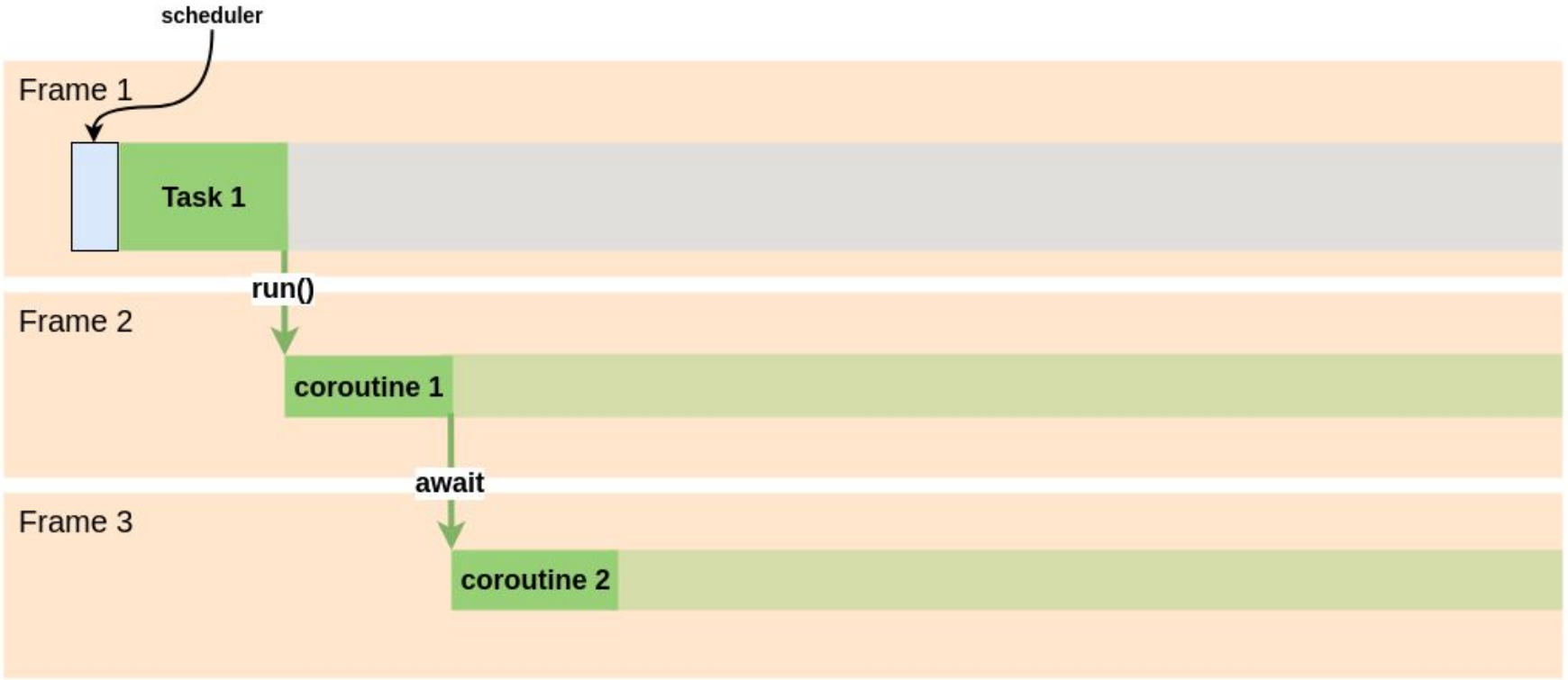
Планировщик внутри event loop проверяет таймеры и запрашивает готовность соединений через системный вызов:

- select или epoll (Linux)
- IOCPs (Windows)

Планировщик запускает обработку
коннекта или таймера для Task 1

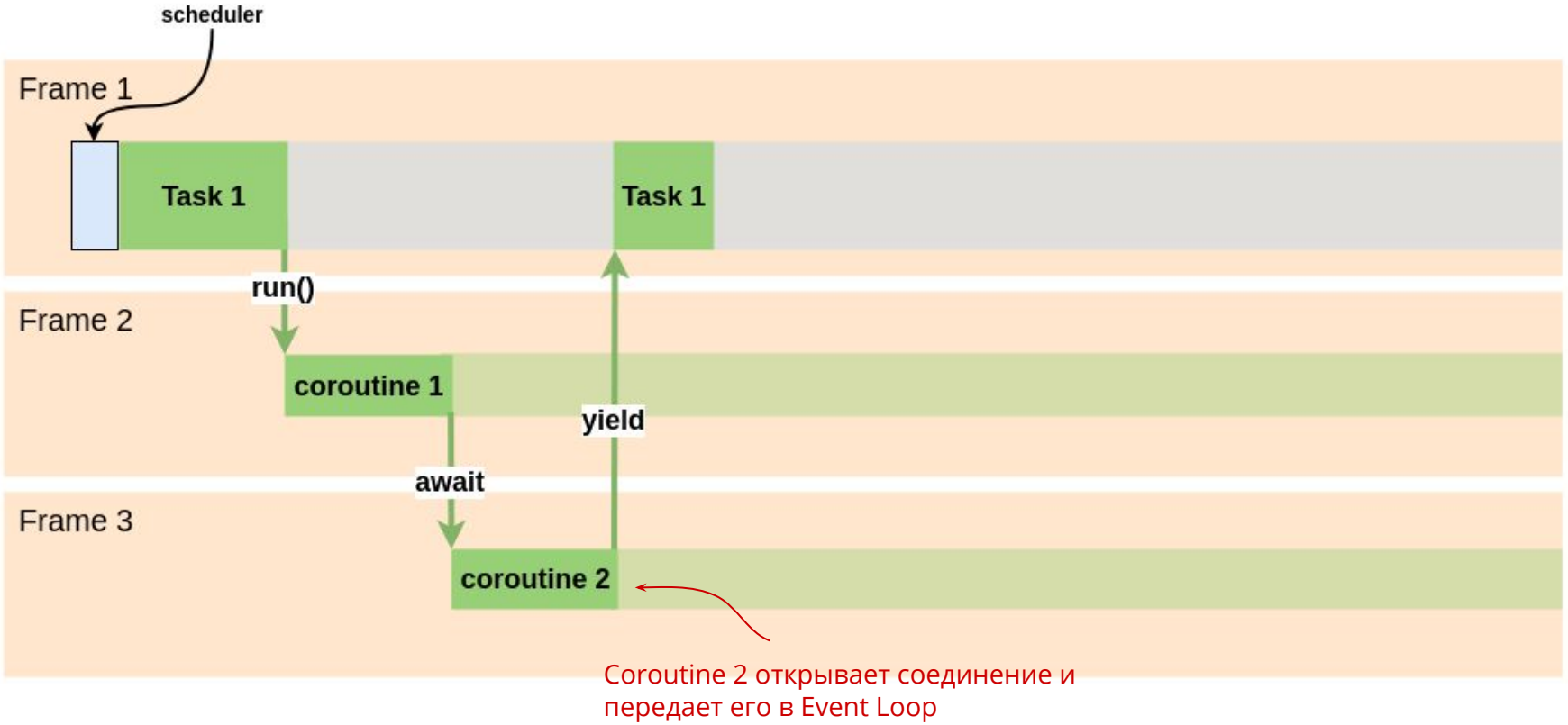

scheduler
Frame 1
Task 1
run()
Frame 2
coroutine 1

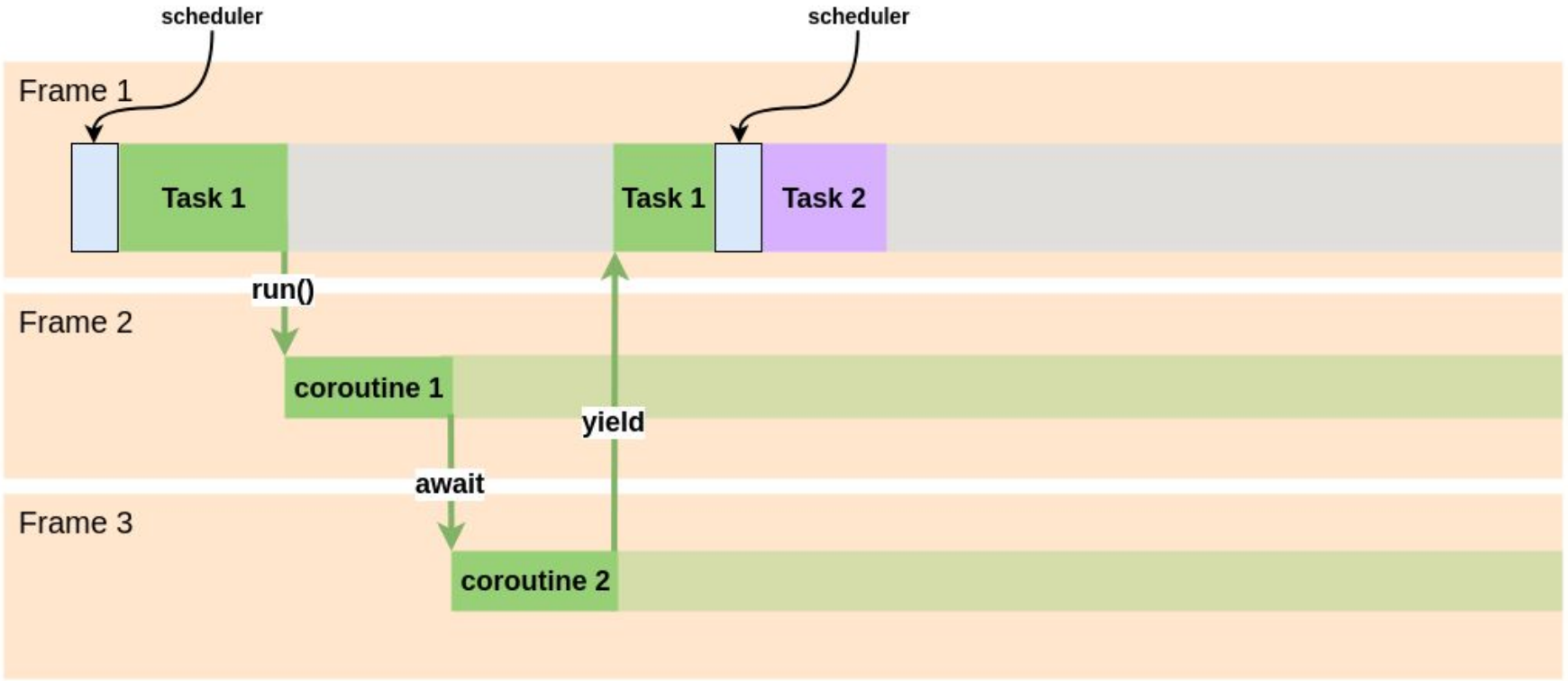
При вызове через await начинает выполняться другая корутина

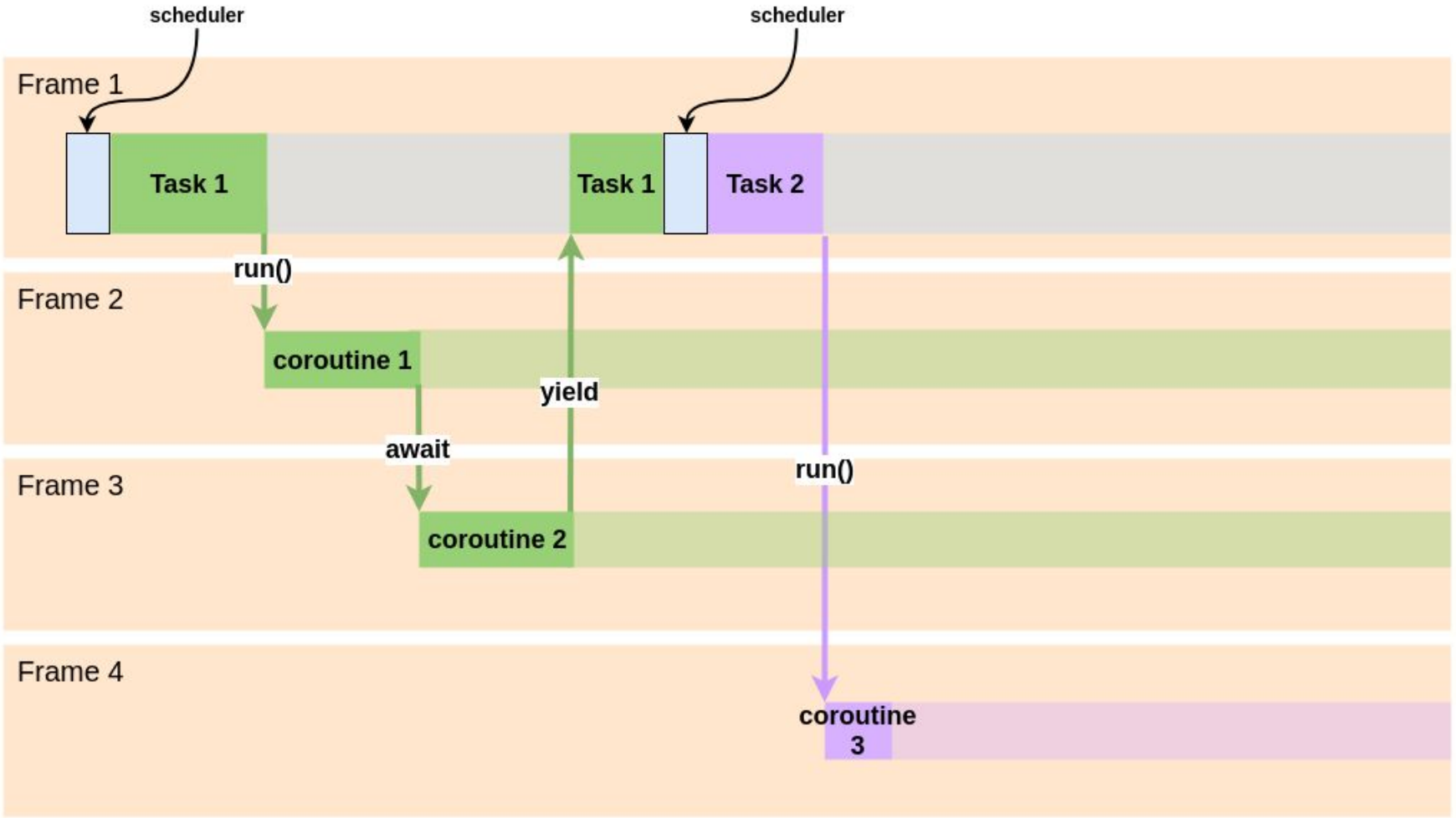

scheduler
Frame 1
Task 1
run()
Frame 2
coroutine 1
await
Frame 3
coroutine 2

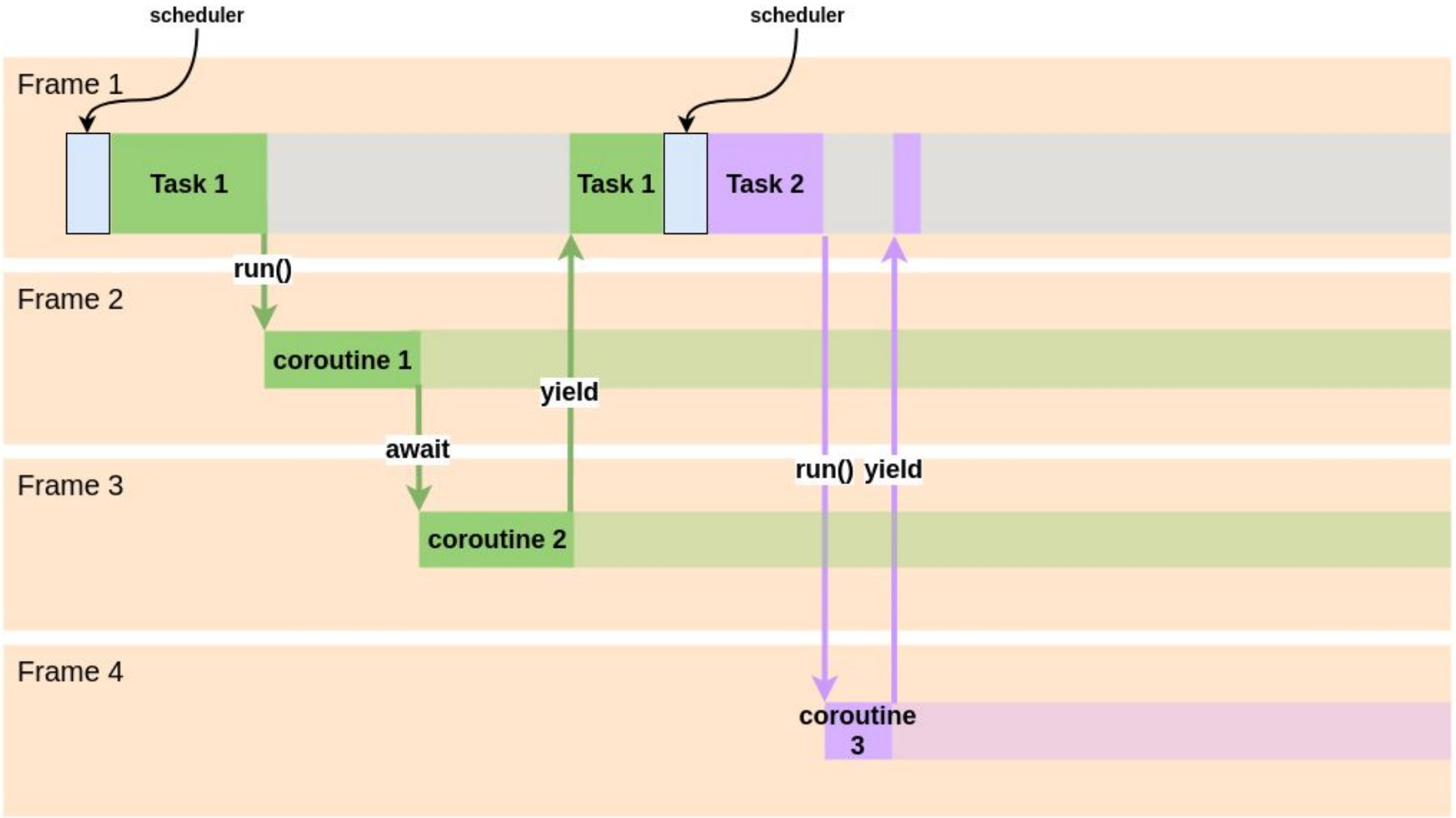
Пользователь может запрограммировать корутину вернуть выполнение:
- в родительскую корутину через return
- в eventloop через asyncio.sleep() (механизм yield)

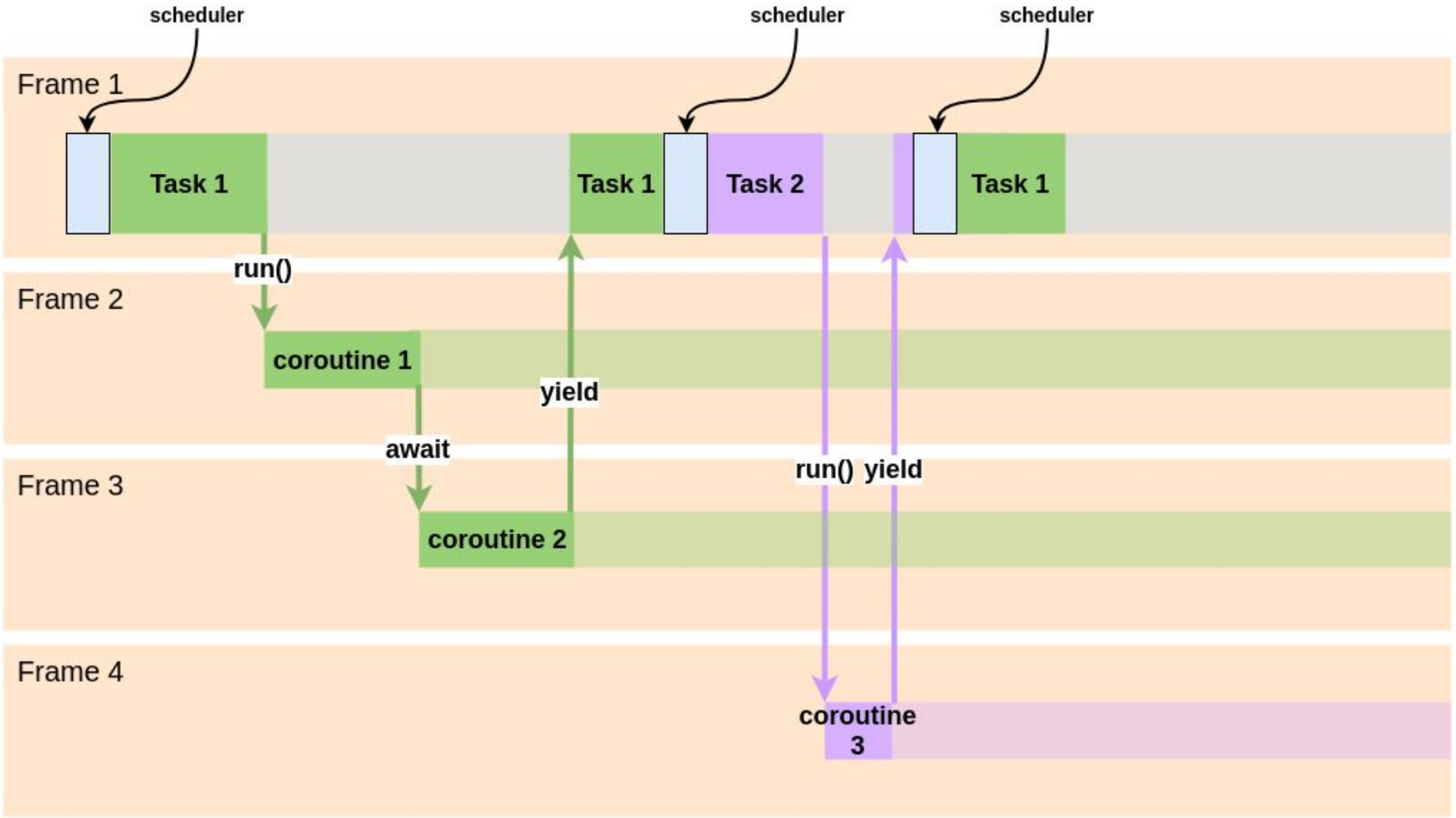
- Внутри asyncio.sleep() задействован механизм yield
- Пользователю нельзя вручную использовать yield в обычных корутинах (где есть return), начиная с версии 3.6
- Корутина с yield — это уже асинхронный генератор

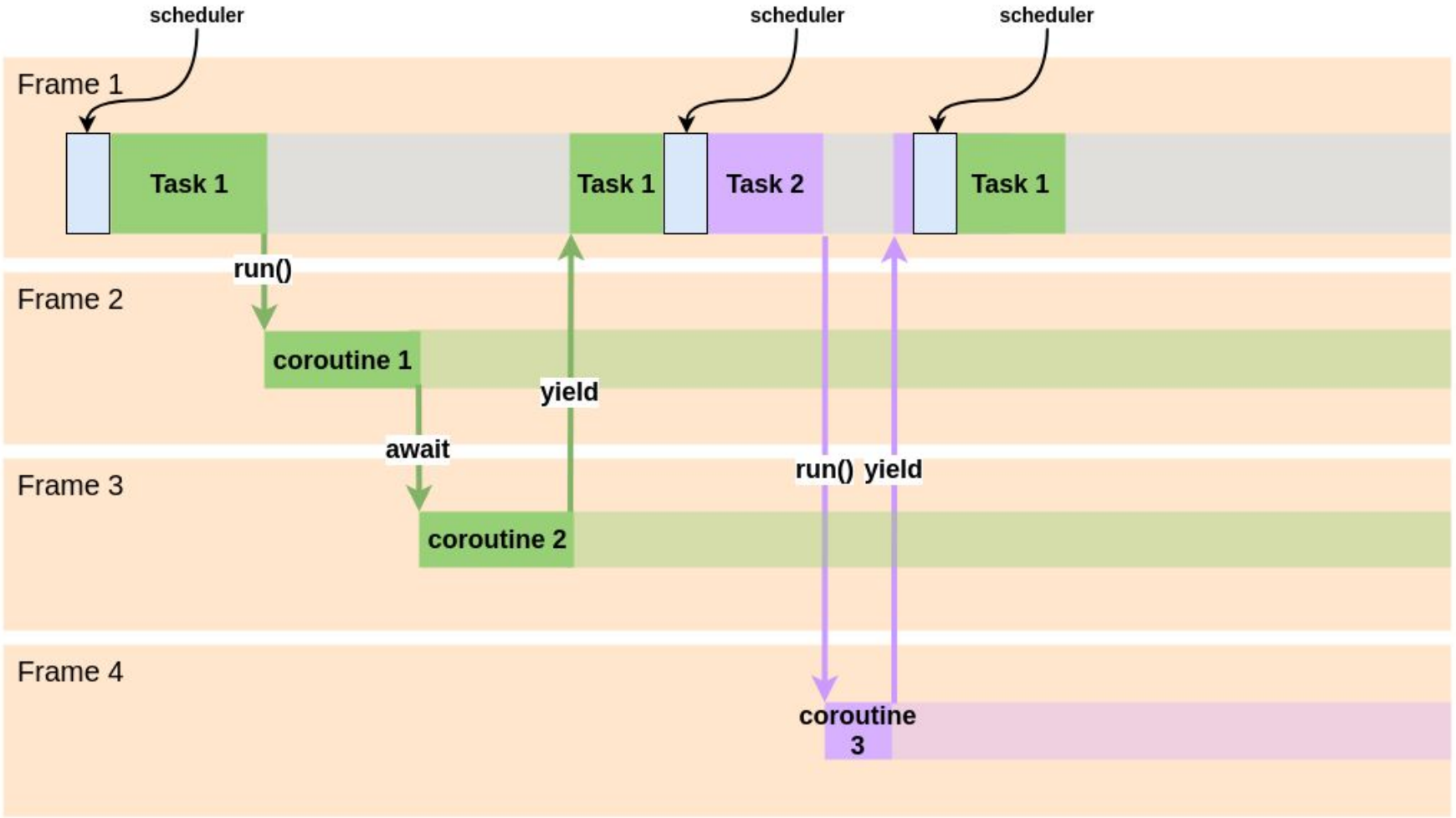

scheduler
Frame 1
Task 1
Task 1
run()
Frame 2
coroutine 1
yield
await
Frame 3
coroutine 2
Coroutine 2 открывает соединение и
передает его в Event Loop

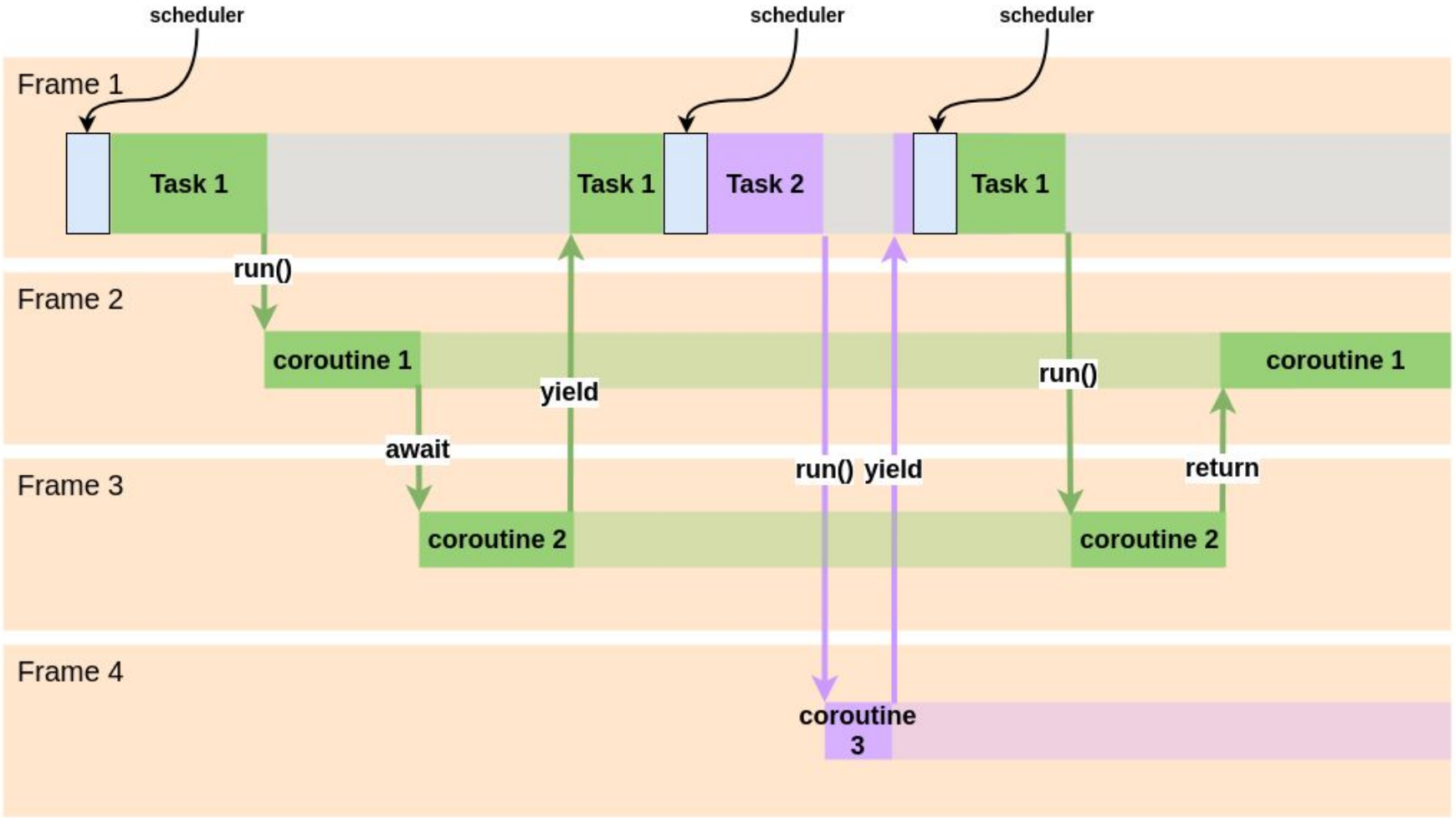

scheduler
scheduler
Frame 1
Task 1
Task 1
Task 2
run()
Frame 2
coroutine 1
yield
await
Frame 3
coroutine 2


scheduler
scheduler
Frame 1
Task 1
Task 1
Task 2
run()
Frame 2
coroutine 1
yield
await
Frame 3
run()
coroutine 2
Frame 4
coroutine
3

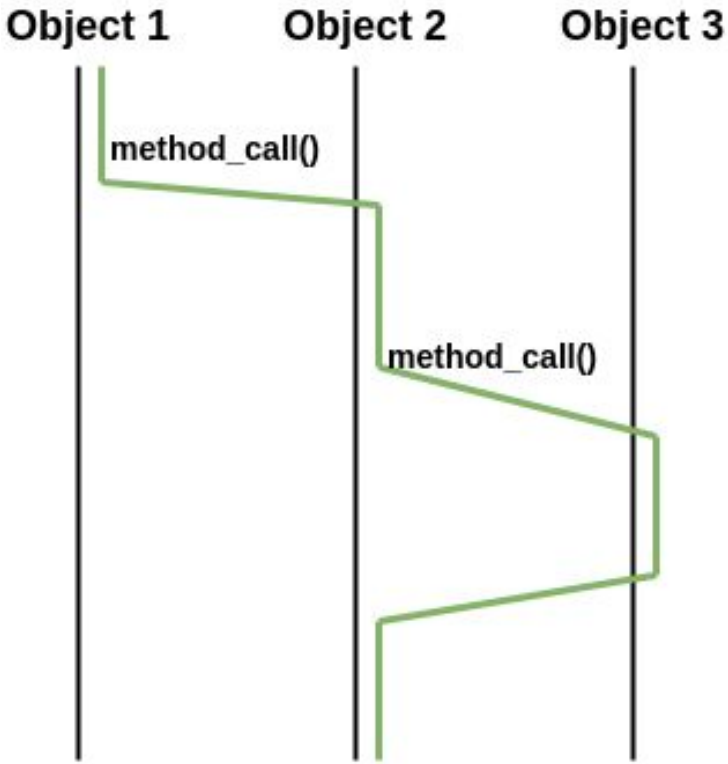

scheduler
scheduler
Frame 1
Task 1
Task 1
Task 2
run()
Frame 2
coroutine 1
yield
await
Frame 3
run()
yield
coroutine 2
Frame 4
coroutine 3


scheduler
scheduler
scheduler
Frame 1
Task 1
Task 1
Task 2
Task 1
run()
Frame 2
coroutine 1
yield
await
Frame 3
run()
yield
coroutine 2
Frame 4
coroutine 3

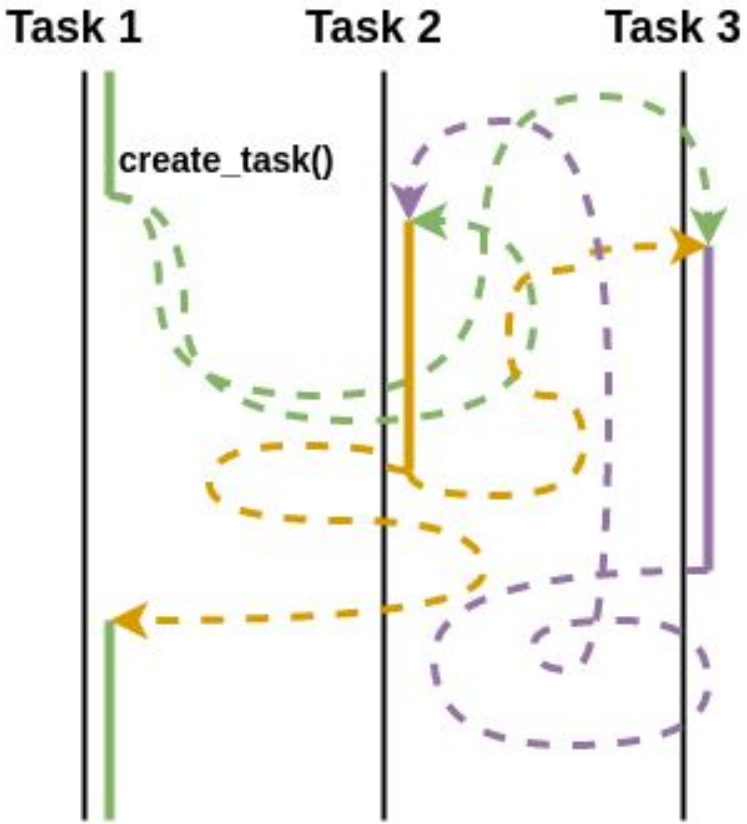

scheduler
scheduler
scheduler
Frame 1
Task 1
Task 1
Task 2
Task 1
run()
Frame 2
coroutine 1
yield
await
Frame 3
run()
yield
coroutine 2
Frame 4
coroutine 3


scheduler
scheduler
scheduler
Frame 1
Task 1
Task 1
Task 2
Task 1
run()
Frame 2
coroutine 1
coroutine 1
yield
run()
await
Frame 3
run()
yield
return
coroutine 2
coroutine 2
Frame 4
coroutine
3

При синхронном стиле программирования порядок выполнения кода понятен


Object 1
Object 2
Object 3
method_call()
method_call()

При использовании asyncio последовательность выполнения кода неочевидна


Task 1
Task 2
Task 3
create_task()

V. Асинхронные библиотеки в питоне

Веб-фреймворки:

- aiohttp
- tornado
- Sanic
- Django 3.0

Асинхронные клиенты:

- aiohttp (http)
- aiopg, asynpg (Postgres)
- Motor, aiomongo (Mongo)
- kafka-python, aiokafka (Kafka)
- aioredis (Redis)

Асинхронные библиотеки ввода-вывода:

- asyncio
- curio
- trio

Event loops:

- asyncio
- uvloop

Будет ли в питоне многопоточность?

PEP-554. Multiple subinterpreters (релиз в 2020)

https://www.python.org/dev/peps/pep-0554/

Спасибо!

Злата Обуховская, Team lead Nvidia,
https://www.facebook.com/zlata.obukhovskaya,
http://bigdatateam.org/,
https://www.facebook.com/bigdatateam/