Способы асинхронной работы Python

Гейне М.А.

20.11.2024

Асинхронное программирование

Асинхронное программирование — концепция программирования, которая заключается в том, что результат выполнения функции доступен не сразу, а через некоторое время в виде некоторого асинхронного (нарушающего обычный порядок выполнения) вызова.

В отличие от синхронного программирования, где компьютер выполняет инструкции последовательно и ожидает завершения системных операций (обращение к устройствам ввода-вывода, жесткому диску, сетевой запрос) блокируя следующие операции в потоке выполнения, в асинхронном программировании длительные операции запускаются без ожидания их завершения и не блокируя дальнейшее выполнение программы.

Global Interpreter Lock (GIL)

Глобальная блокировка интерпретатора (GIL) - это мьютекс, который защищает доступ к объектам Python в CPython, обеспечивая одновременное выполнение байткода Python только одним потоком даже на многоядерных системах. GIL упрощает управление памятью, особенно для системы подсчета ссылок в CPython, но ограничивает истинную многопоточную производительность для задач, привязанных к процессору

Concurrency vs. Parallelism

Аспект	Concurrency	Parallelism
Модель исполнения	Пересекающиеся задачи (не обязательно одновременные)	Задания выполняются одновременно
Фокус	Эффективное управление множеством задач	Ускорение выполнения задач с использованием большего количества ресурсов
Требования	Не требует нескольких ядер	Требуется несколько ядер/ процессоров
Примеры использования	Обработка задач, связанных с вводом-выводом, таких как web-scrapping или чат-серверы	Выполнение задач, связанных с процессором, например, обработка данных или моделирование

Процессы и потоки

- Процесс - это независимая единица выполнения с собственным пространством памяти. Каждый процесс выполняется в своей изолированной среде
- Поток - это малая единица выполнения внутри процесса. Потоки в одном процессе используют одно и то же пространство памяти

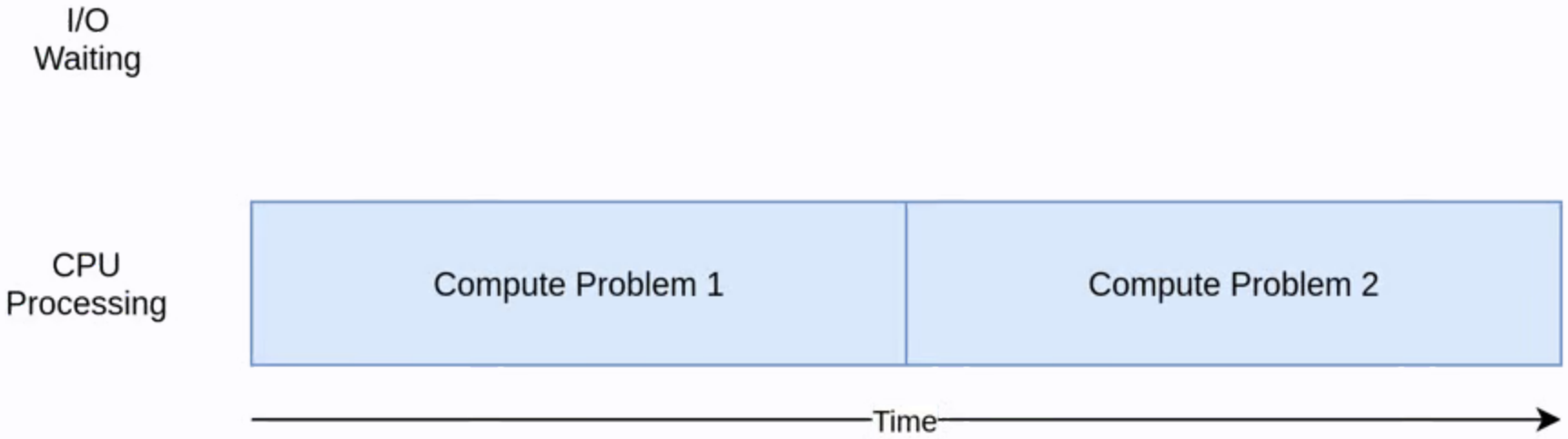
Аспект	Процессы	Потоки
Память	Отдельное пространство памяти для каждого процесса	Разделяемая память в рамках одного процесса
Связь	Необходимы явные механизмы	Разделенная память обеспечивает прямой доступ
Нагрузка на производительность	Большая нагрузка (из-за изоляции)	Меньшая нагрузка
Влияние сбоя	Изолированный - сбой одного процесса не влияет на другие	Общий - сбой может повлиять на все потоки
Параллелизм	Настоящий параллелизм возможен на многоядерных процессорах	Ограничен глобальной блокировкой интерпретатора (GIL) в Python
Подходит для	CPU-ограниченные задачи, требующие изоляции	I/O-ограниченные задачи, требующие общего состояния

В Python возможно использовать как многопоточность, так и мультипроцессность, однако наличие GIL вводит ряд ограничений:

- Если в программе задействуется несколько потоков, выполняться сможет только один из них одновременно
- Задачи, связанные с I/O могут получить прирост к производительности с использованием нескольких потоков, т.к. во время ожидания данных на одном потоке, другой может исполняться
- Задачи, связанные с вычислениями на CPU, не могут быть распределены между потоками из-за GIL
- Задачи, связанные с вычислениями на CPU, возможно решать в нескольких процессах, минуя ограничения GIL

I/O-ограниченные задачи

CPU-ограниченные задачи



Межпроцессное взаимодействие (IPC)

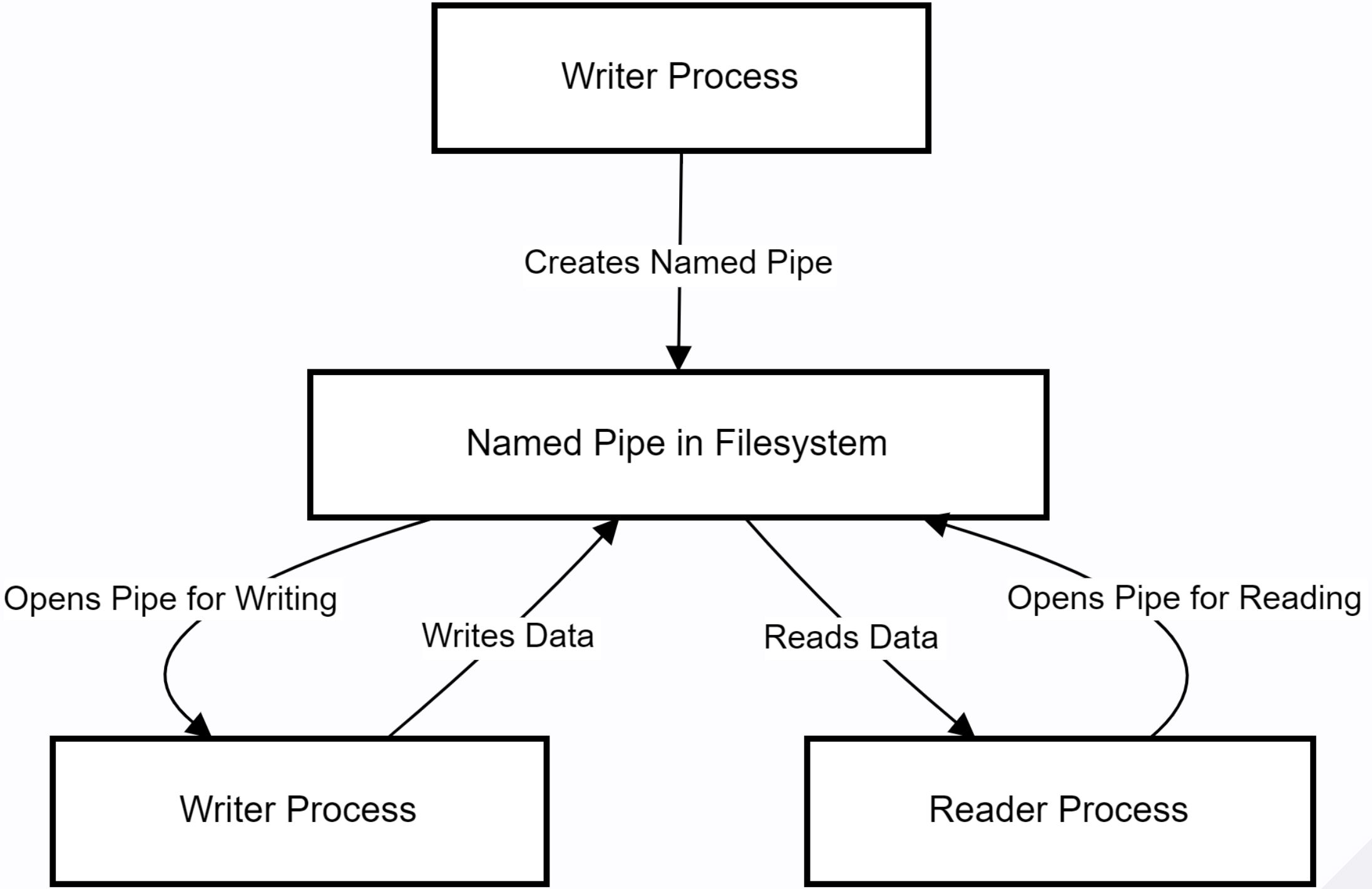
- Файлы
- Общая память
- Передача сообщений
- Анонимные и именованные каналы
- Сокеты
- Сигналы
- Общие объекты

Анонимные каналы

- Анонимные каналы - это **однонаправленные** каналы связи, которые существуют только в течение времени жизни использующих их процессов. Они не имеют имен и обычно используются между родительским процессом и его дочерними процессами
- Один процесс пишет в канал, другой процесс из него читает

Именованные каналы

- Именованные каналы идентифицируются именем в файловой системе. Они обеспечивают связь между несвязанными процессами, и канал сохраняется до тех пор, пока не будет явно удалён
- Двунаправленные (в большинстве реализаций, хотя встречаются и однонаправленные)
- Существует как файл в файловой системе до тех пор, пока не будет удален
- Позволяет взаимодействовать между несвязанными процессами


Writer Process
Creates Named Pipe
Named Pipe in Filesystem
Opens Pipe for Writing
Writes Data
Reads Data
Opens Pipe for Reading
Writer Process
Reader Process

Сокеты

Сокеты - это фундаментальная концепция взаимодействия между процессами, часто по сети. Они предоставляют процессам механизм для отправки и получения данных как на одной машине, так и на разных машинах. Сокеты могут использовать различные протоколы, например, TCP или UDP

1. **Создание сокета** на сервере и клиенте
2. **Привязывание** сокета на сервере к IP и порту
3. Сервер **прослушивает** сокет на наличие подключений
4. Клиент **подключается** к серверу по IP и порту
5. Сервер и клиент **обмениваются данными**
6. После обмена, сокеты **закрываются**

```python
import socket

# Create a socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('localhost', 5000))
server_socket.listen(1)

print("Server is listening...")
conn, addr = server_socket.accept()
print(f"Connection from {addr}")

# Receive and send data
data = conn.recv(1024)
print(f"Received: {data.decode()}")
conn.sendall(b"Hello, client!")

conn.close()
server_socket.close()
```

```python
import socket

# Create a socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('localhost', 5000))

# Send and receive data
client_socket.sendall(b"Hello, server!")
data = client_socket.recv(1024)
print(f"Received: {data.decode()}")

client_socket.close()
```

Состояние гонки и взаимной блокировки

Состояние **гонки** возникает, когда два или более потока или процесса получают доступ к общим ресурсам (например, переменным, файлам или памяти) и пытаются изменить их одновременно, что приводит к непредсказуемому поведению. Результат зависит от **времени** или последовательности выполнения. Это происходит потому, что потоки или процессы "мчатся", пытаясь выполнить свои задачи, часто без надлежащей синхронизации

Взаимная блокировка возникает, когда два или более потока или процесса застревают, ожидая друг от друга освобождения ресурсов, которые им необходимы для продолжения работы. Это приводит к неопределённой остановке выполнения. Процессы **блокируются навсегда**. Обычно происходит, когда несколько блокировок используются и занимаются в разном порядке

Способы синхронизации

- Мьютексы - организуют выделенный, единичный доступ к ресурсу
- Семафоры - организуют доступ к ресурсу нескольким потокам единовременно
- События - механизм сообщения от одного процесса другому о некотором событии
- Условия - потоки могут дожидаться выполнения условия, после чего продолжить исполнение
- Барьеры - блокировка нескольких процессов до тех пор, пока все из них не достигнут барьера, после чего исполнение продолжится
- Очереди - структура данных, позволяющая одному процессу добавлять данные, а другому - забирать

I/O-ограниченная задача в Python

Рассмотрим пример следующей задачи: скачать несколько веб-страниц по их URL. Для этого потребуется многократно обращаться к внешним ресурсам

Синхронное решение

```
import requests
import time


def download_site(url, session):
    with session.get(url) as response:
        print(f"Read {len(response.content)} from {url}")


def download_all_sites(sites):
    with requests.Session() as session:
        for url in sites:
            download_site(url, session)


if __name__ == "__main__":
    sites = [
        "https://www.jython.org",
        "http://olympus.realpython.org/dice",
    ] * 80
    start_time = time.time()
    download_all_sites(sites)
    duration = time.time() - start_time
    print(f"Downloaded {len(sites)} in {duration} seconds")
```

Плюсы решения:

- простое, легко отладить и понять

Минусы решения:

```
$ ./io_non_concurrent.py
   [most output skipped]
Downloaded 160 in 14.289619207382202 seconds
```

Решение через `threading`

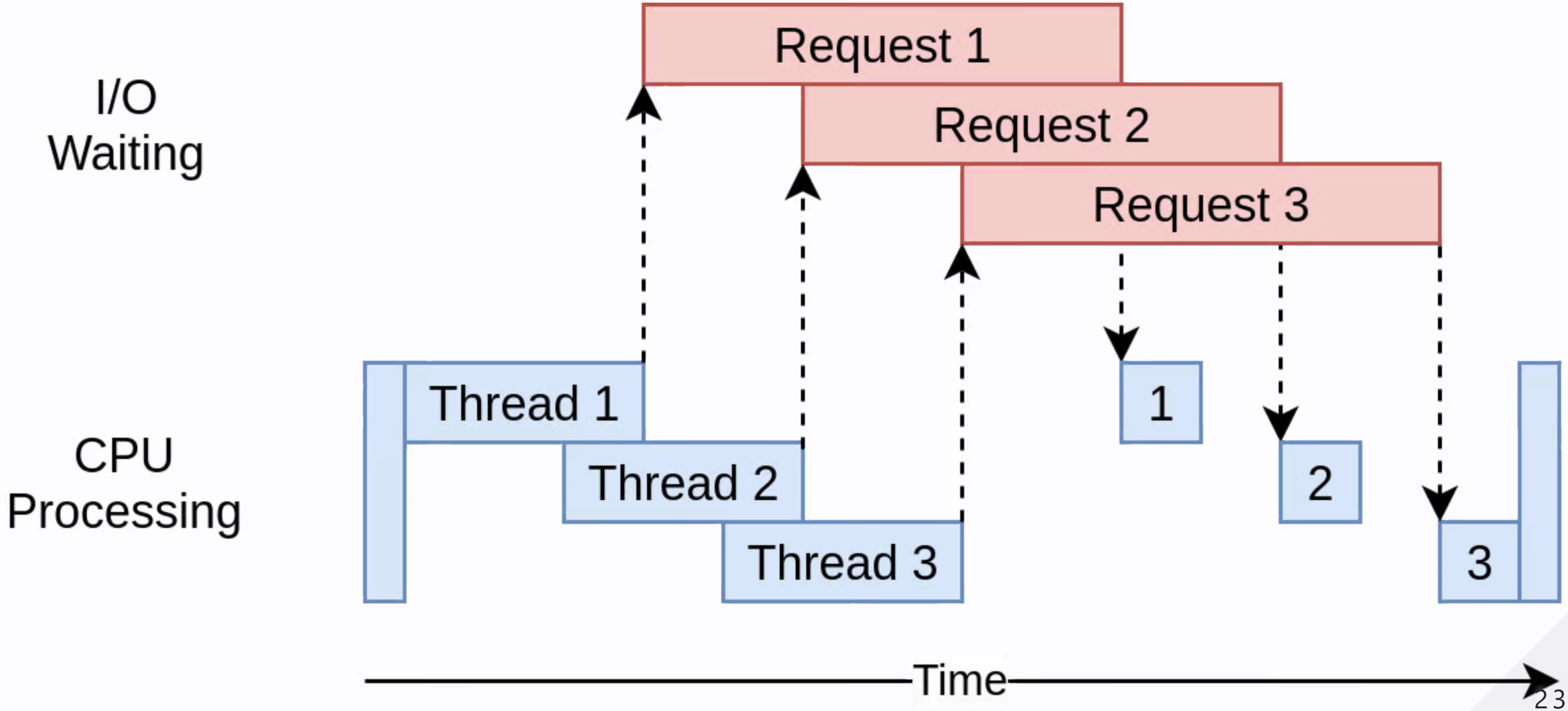
Модуль `threading` предоставляет возможность запускать несколько потоков в рамках одного процесса. Потоки совместно используют одно и то же пространство памяти. Потоки обеспечивают параллельное выполнение задач, но ограничены **Global Interpreter Lock (GIL)**

```python
import concurrent.futures
import requests
import threading
import time


thread_local = threading.local()


def get_session():
    if not hasattr(thread_local, "session"):
        thread_local.session = requests.Session()
    return thread_local.session


def download_site(url):
    session = get_session()
    with session.get(url) as response:
        print(f"Read {len(response.content)} from {url}")

def download_all_sites(sites):
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        executor.map(download_site, sites)
```

Следует обратить внимание:

1. `ThreadPoolExecutor` выделяет ограниченный набор потоков и исполнителей. Распределение задач между потоками и управление ими берут на себя исполнители
2. Потоки управляются не только исполнителями, но и ОС, из-за чего они могут быть приостановлены в любой момент
3. Каждому потоку необходима `requests.Session()`, которая не является потокобезопасной. Для решения этой проблемы задействуется `threading.local()` - собственное хранилище потоков. `local()` создаётся один раз, но при этом для каждого потока содержание отличается
4. В решении используется 5 потоков. Количество потоков следует подбирать так, чтобы было ускорение от распределения задач, которое бы не нейтрализовалось дополнительной работой по управлению потоками

```
$ ./io_threading.py
   [most output skipped]
Downloaded 160 in 3.7238826751708984 seconds
```



Решение через `asyncio`

`asyncio` - это библиотека для **асинхронного программирования**, использующая однопоточную модель цикла событий. Задачи не выполняются параллельно, а выполняются совместно для достижения максимальной эффективности. В нем используются **корутины** (`async def`), которые передают управление циклу событий во время операций ввода-вывода или других ожидающих операций

Модель работы `asyncio`

- Один объект Python, называемый циклом событий, управляет тем, как и когда запускается каждая задача
- Цикл событий знает о каждой задаче и знает, в каком состоянии она находится
- Состояние готовности - задаче нужно выполнить свою работу и готова к запуску; состояние ожидания - задача ожидает завершения какого-то внешнего действия
- Два списка задач, по одному для каждого из этих состояний
- Цикл выбирает одну из готовых задач и запускает ее в работу
- Эта задача выполняется самостоятельно до тех пор, пока она не передаст управление обратно циклу событий
- Задача помещается в нужный список в соответствии с её состоянием
- Задачи в списке ожидания проверяются на готовность к исполнению и переводятся в список готовых задач по необходимости
- Цикл выбора задачи повторяется
- Важно: задачи преедают управление самостоятельно и не могут быть внезапно прерваны, что упрощает управление ресурсами

`async` и `await`

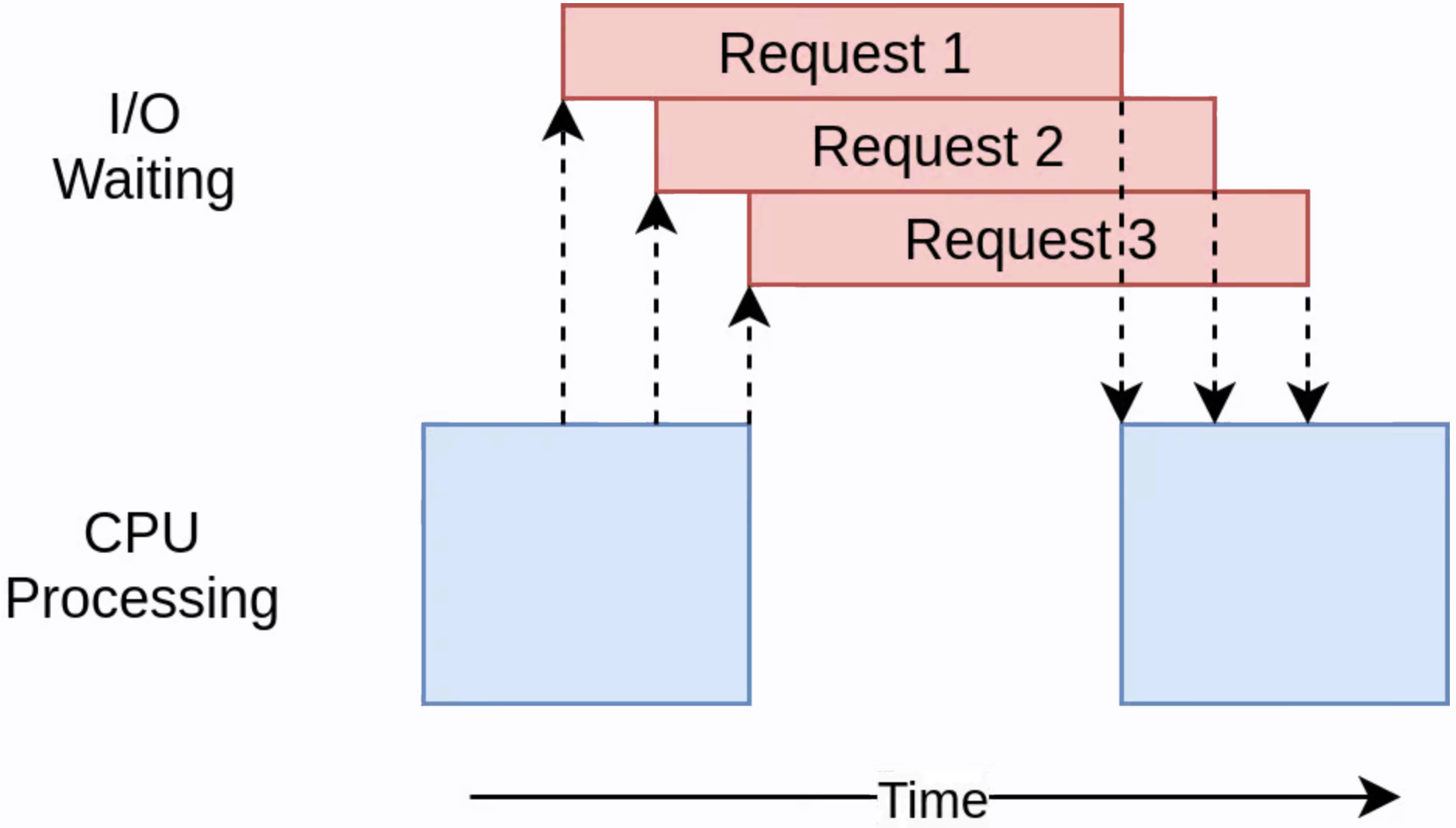
- `await` - способ вернуть контроль обратно в цикл событий и показать, что задача переводится в режим ожидания
- `async` показывает, что следующий блок кода асинхронный и может переходить в режим ожидания

```python
import asyncio
import time
import aiohttp


async def download_site(session, url):
    async with session.get(url) as response:
        print("Read {0} from {1}".format(response.content_length, url))


async def download_all_sites(sites):
    async with aiohttp.ClientSession() as session:
        tasks = []
        for url in sites:
            task = asyncio.ensure_future(download_site(session, url))
            tasks.append(task)
        await asyncio.gather(*tasks, return_exceptions=True)


if __name__ == "__main__":
    sites = [
        "https://www.jython.org",
        "http://olympus.realpython.org/dice",
    ] * 80
    start_time = time.time()
    asyncio.get_event_loop().run_until_complete(download_all_sites(sites))
    duration = time.time() - start_time
    print(f"Downloaded {len(sites)} sites in {duration} seconds")
```

Следует обратить внимание:

- Т.к. программа теперь выполняется в одном потоке, сессия может быть единой
- `asyncio.ensure_future()` создаёт и запускает задачи
- `asyncio.gather()` поддерживает контекст до тех пор, пока все задачи не будут выполнены
- `asyncio.get_event_loop().run_until_complete()` создаёт цикл событий и дожидается исполнения всех задач в нём
- С версии Python 3.7 `asyncio.get_event_loop().run_until_complete()` -> `asyncio.run()`

```
$ ./io_asyncio.py
   [most output skipped]
Downloaded 160 in 2.5727896690368652 seconds
```



`asyncio` имеет ряд преимуществ: высокая производительность, хорошая скалируемость, более простое дееление ресурсов и явное понимание точек смены контекста

Однако главной проблемой является необходимость использовать специальные версии библиотек с поддержкой `asyncio`. К примеру, `requests` не может уведомлять о том, что ожидает ресурс и временно заблокирован, поэтому в примере используется `aiohttp`

Ещё одной проблемой является сама модель кооперативной обработки. Каждая задача должна сообщать, когда она находится в ожидании и отдавать управление другим задачам. Если допущена ошибка и какая-то задача отказывается отдать управление, другие задачи исполняться не будут

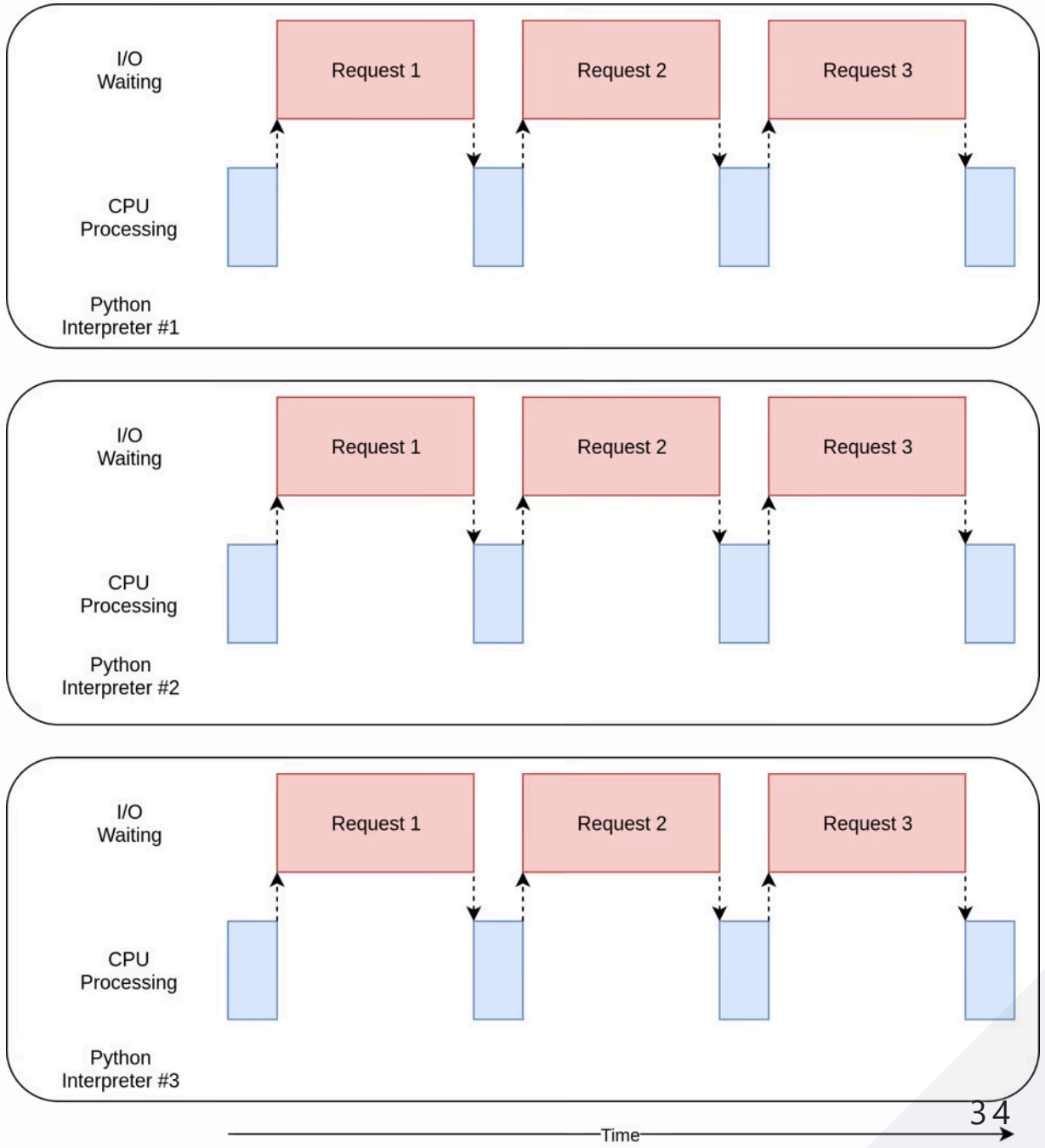
Решение через `multiprocessing`

Модуль `multiprocessing` позволяет программам порождать несколько **процессов**, каждый со своим собственным интерпретатором Python и пространством памяти. Это позволяет обойти GIL, обеспечивая истинное параллельное выполнение. Использует несколько процессов, которые могут работать одновременно на нескольких ядрах процессора

```
import requests
import multiprocessing
import time

session = None


def set_global_session():
    global session
    if not session:
        session = requests.Session()


def download_site(url):
    with session.get(url) as response:
        name = multiprocessing.current_process().name
        print(f"{name}:Read {len(response.content)} from {url}")


def download_all_sites(sites):
    with multiprocessing.Pool(initializer=set_global_session) as pool:
        pool.map(download_site, sites)
```

- `multiprocessing.Pool` создает несколько отдельных процессов интерпретатора Python и заставляет каждый из них выполнять указанную функцию над некоторыми элементами в итераторе
- Сообщение между процессами организовано модулем `multiprocessing`
- По умолчанию `multiprocessing.Pool()` определит количество процессоров в вашем компьютере и создаст соотвтетсвующее количество процессов
- Каждый процесс имеет собственную память, следовательно требует собственную сессию. Чтобы не пересоздавать сессию для каждого запроса, `initializer=set_global_session` создаст глобальную переменную в каждом процессе, которая будет сохраняться между вызовами функции

```
$ ./io_mp.py
    [most output skipped]
Downloaded 160 in 5.718175172805786 seconds
```



CPU-ограниченная задача в Python

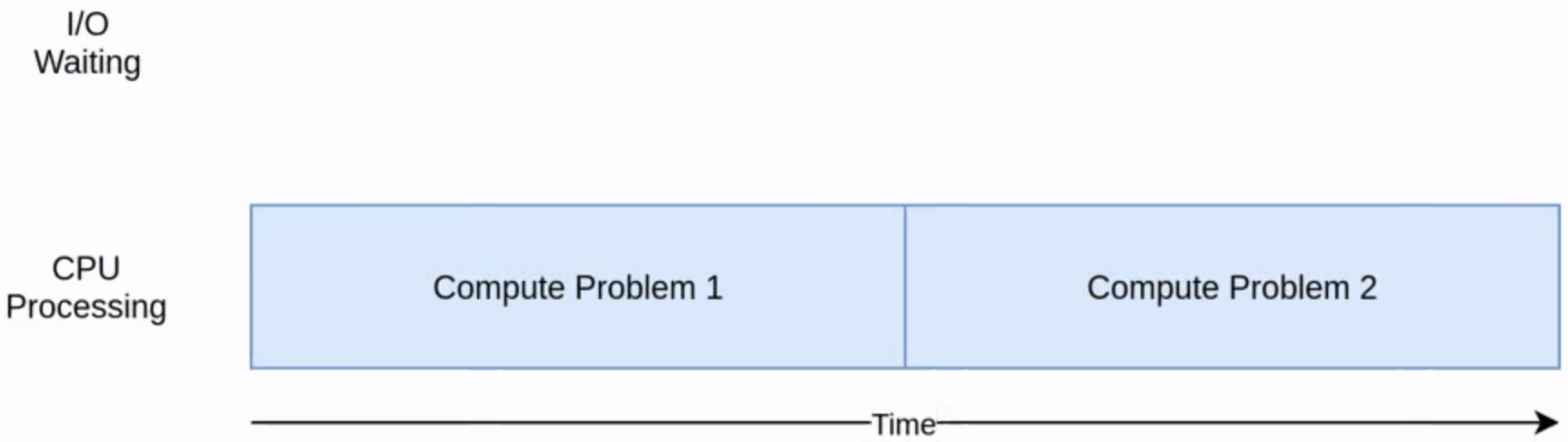
Рассмотрим пример следующей задачи: рассчитать сумму квадратов всех чисел от 0 до n

```
def cpu_bound(number):
    return sum(i * i for i in range(number))
```

Синхронная версия

```
import time


def cpu_bound(number):
    return sum(i * i for i in range(number))


def find_sums(numbers):
    for number in numbers:
        cpu_bound(number)


if __name__ == "__main__":
    numbers = [5_000_000 + x for x in range(20)]

    start_time = time.time()
    find_sums(numbers)
    duration = time.time() - start_time
    print(f"Duration {duration} seconds")
```

```
$ ./cpu_non_concurrent.py
Duration 7.834432125091553 seconds
```



Решение через `threading` и `asyncio`

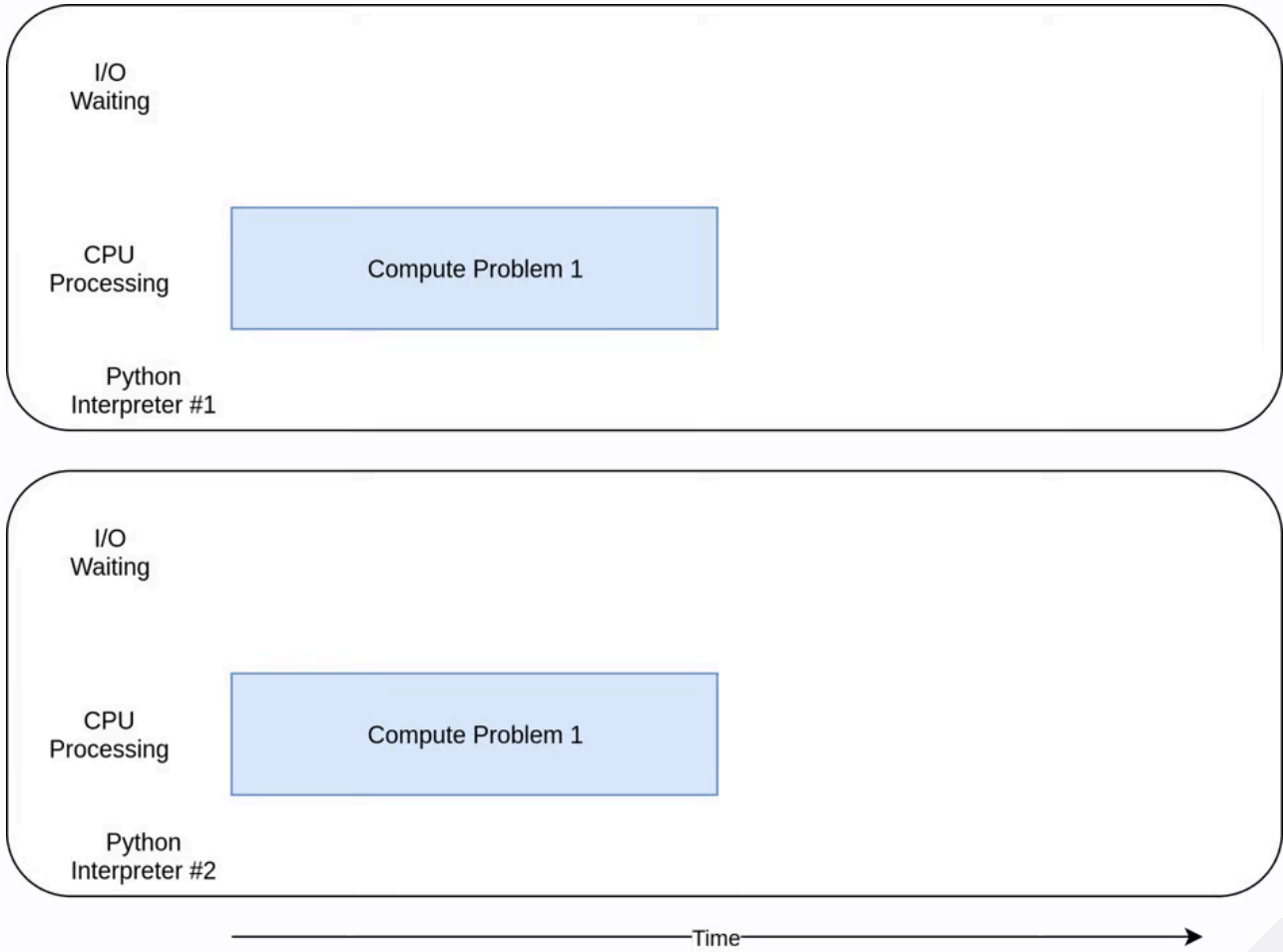
```
$ ./cpu_threading.py
Duration 10.407078266143799 seconds
```

Время решения задачи увеличилось, т.к. добавилось время на организацию потоков или цикла событий, но задача всё так же исполнялась в одном потоке и полностью использовала мощности 1 ядра процессора

Решение через `multiprocessing`

```
import multiprocessing
import time


def cpu_bound(number):
    return sum(i * i for i in range(number))


def find_sums(numbers):
    with multiprocessing.Pool() as pool:
        pool.map(cpu_bound, numbers)


if __name__ == "__main__":
    numbers = [5_000_000 + x for x in range(20)]

    start_time = time.time()
    find_sums(numbers)
    duration = time.time() - start_time
    print(f"Duration {duration} seconds")
```

```
$ ./cpu_mp.py
Duration 2.5175397396087646 seconds
```



Итог

- Для задач, в которых много времени уходит на ожидание ресурсов, следует организовать многопоточную или кооперативную работу с использованием `threading` или `asyncio`
- Для задач, в которых основной нагрузкой являются вычисления на процессоре, следует организовать многопроцессную обработку с использованием `multiprocessing`

Python 3.13 и GIL

В Python 3.13 появилась экспериментальная возможность отключить GIL. Вместо GIL привносятся новые методы управления памятью, подсчёта ссылок и работы с контейнерами. Отсутствие GIL позволяет задействовать действительную многопоточность с параллельным их исполнением. Так возрастает производительность многопоточных программ. Однако есть и недостатки: падает производительность однопоточных и многопроцессных программ, а также теряется совместимость с библиотеками

Спасибо за внимание :)