Дипломная работа по теме:

# *Сравнение различных подходов к реализации асинхронного программирования: asyncio, threading и multiprocessing*

Автор: **Портнов Дмитрий Борисович**

# Оглавление

# Введение

## Обоснование выбора темы

- Описание проблемы:
  Компьютерные программы часто имеют дело с длительными процессами. Например: получают данные из базы, ожидают ответа от интернет-сервера или производят сложные вычисления. Пока выполняется одна операция, можно было бы завершить еще несколько. А бездействие приводит к снижению продуктивности и убыткам. Асинхронный подход увеличивает эффективность, потому что позволяет не блокировать основной поток выполнения программы и производить другие доступные операции
- История вопроса:
  Начиная с операционных систем, которые были выпущены после MS-DOS (UNIX, Linux, OS/2, MS Windows и другие), многозадачность стала доступна на уровне системы. В это же время стали доступны паттерны асинхронного программирования в основных алгоритмических языках, таких как C и C++, но массовая реализация их в практических приложениях началась значительно позже. На начальном этапе общая многозадачность достигалась за счет асинхронности выполнения приложений, которая определялась операционной системой и которая не учитывала особенности алгоритмики самих приложений
- Практические потребности:
  В настоящее время ввиду большой конкуренции на рынке программного обеспечения и повышенным требованиям к оперативности выполнения алгоритмов, владение техникой асинхронного программирования выходит на передний план в ряду компетенций современного программиста. Одной из особенностей асинхронного подхода в программировании является более ответственный подход к проектированию алгоритмов и использованию моделей данных
- Личный интерес и перспективы использования:
  Использование асинхронных моделей программирования для создания ИТ-продуктов позволяет иметь конкурентные преимущества на рынке труда, что открывает перспективы для карьерного роста и развития.

## Цели и задачи исследования

- Определить и спроектировать основные алгоритмические конструкции, на которых будут применены паттерны асинхронного программирования
- Разработать программную реализацию каждого паттерна для каждой алгоритмической конструкции
- Провести сравнительный анализ времени выполнения и используемых ресурсов каждого из паттернов на каждой алгоритмической модели
- Выявить, какие из асинхронных паттернов являются наилучшими с точки зрения времени выполнения и используемых ресурсов для каждой алгоритмической конструкции

# Основные понятия и определения

## Синхронность и асинхронность

При синхронных операциях задачи выполняются синхронно, одна за другой. При асинхронных операциях задачи могут запускаться и завершаться независимо друг от друга. Одна асинхронная задача может стартовать и продолжать выполняться, пока выполнение переходит к новой задаче. Асинхронные задачи не блокируют (не заставляют ждать завершения) операции и обычно выполняются в фоновом режиме. Например, вам нужно позвонить в туристическое агентство, чтобы забронировать билеты на следующий отпуск. А перед тем, как отправиться в тур, вам нужно

отправить электронное письмо своему начальнику. В синхронном режиме вы сначала звоните в туристическое агентство, если вас на минуту переключают на ожидание, вы продолжаете ждать и ждать. Когда все будет готово, вы начнете писать письмо своему боссу. Вот так вы выполняете одну задачу за другой. Но если проявить смекалку и во время ожидания начать писать письмо, то, когда с вами заговорят, вы приостановите написание письма, поговорите с ними, а затем продолжите писать письмо. Вы также можете попросить друга сделать звонок, пока вы дописываете письмо. Это и есть асинхронность. Задачи не блокируют друг друга.

### Конкуренция и параллелизм

Конкуренция подразумевает, что две задачи выполняются вместе. В нашем предыдущем примере, когда мы рассматривали асинхронный пример, мы одновременно выполняли звонок турагенту и писали письмо. Это и есть конкуренция. Когда мы говорили о том, что нам поможет друг со звонком, в этом случае обе задачи выполнялись бы параллельно. Параллелизм — это, по сути, форма конкуренции. Но параллелизм зависит от аппаратного обеспечения. Например, если в процессоре только одно ядро, две операции не могут выполняться параллельно. Они просто делят временные срезы одного и того же ядра. Это конкуренция, но не параллельность. Но когда у нас есть несколько ядер, мы можем выполнять две или более операций (в зависимости от количества ядер) параллельно.

### Краткие итоги

Вот краткие итоги вышесказанного:

- **Синхронность**: Блокирование операций.
- **Асинхронность**: Неблокирующие операции.
- **Конкуренция**: Совместное выполнение.
- **Параллелизм**: Параллельное выполнение задач.

Параллелизм подразумевает конкуренцию, но конкуренция не всегда означает параллелизм.

## Теоретическая часть

### Потоки и процессы

В Python уже очень давно существуют потоки, которые позволяют выполнять операции параллельно. Но существовала и существует проблема с глобальной блокировкой интерпретатора (GIL), для которой потоки не могут обеспечить истинный параллелизм. Однако с появлением многопроцессорности появилась возможность задействовать несколько ядер в Python.

### Потоки (threading)

Давайте рассмотрим небольшой пример. В следующем коде рабочая функция будет выполняться в нескольких потоках, асинхронно и параллельно. После этого, для сравнения, тот же самый код будет выполнен синхронно. В результатах будет выведено время выполнения для обоих случаев:

```
import threading
import time

start_time = time.time()
maxnumber = 10

def worker(number):
    sleep = maxnumber - number
    time.sleep(sleep)
    print("I am Worker {}, I slept for {} seconds".format(number, sleep))


#------------- Async -------------
```

```
print("Async is started, let's see when it finish!")
threads = []
for i in range(maxnumber):
    thread = threading.Thread(target=worker, args=(i,))
    thread.start()
    threads.append(thread)

print("All Threads are queued")

for thread in threads:
    thread.join()

print("Async is finished. Elapced:", time.time() - start_time)
#------------ Sync -------------
print("Sync is started, let's see when it finish!")

start_time = time.time()
for i in range(maxnumber):
    worker(i)

print("Sync is finished. Elapced:", time.time() - start_time)
```

Результат выполнения:

```
Async is started, let's see when it finish!
All Threads are queued
I am Worker 9, I slept for 1 seconds
I am Worker 8, I slept for 2 seconds
I am Worker 7, I slept for 3 seconds
I am Worker 6, I slept for 4 seconds
I am Worker 5, I slept for 5 seconds
I am Worker 4, I slept for 6 seconds
I am Worker 3, I slept for 7 seconds
I am Worker 2, I slept for 8 seconds
I am Worker 1, I slept for 9 seconds
I am Worker 0, I slept for 10 seconds
Async is finished. Elapced: 9.99811840057373
Sync is started, let's see when it finish!
I am Worker 0, I slept for 10 seconds
I am Worker 1, I slept for 9 seconds
I am Worker 2, I slept for 8 seconds
I am Worker 3, I slept for 7 seconds
I am Worker 4, I slept for 6 seconds
I am Worker 5, I slept for 5 seconds
I am Worker 6, I slept for 4 seconds
I am Worker 7, I slept for 3 seconds
I am Worker 8, I slept for 2 seconds
I am Worker 9, I slept for 1 seconds
Sync is finished. Elapced: 54.98807501792908
```

Видно, что мы запускаем 10 потоков, они выполняют работу вместе. Когда мы запускаем потоки (и тем самым выполняем рабочую функцию), операция не ждет завершения потоков, прежде чем перейти к следующему оператору печати. Таким образом, это асинхронная операция, которая

выполняется в 5 раза быстрее, чем синхронная. В данном примере мы передали конструктору Thread функцию, но, при желании мы могли бы создать подкласс и реализовать код в виде метода (более близко к ООП).

## Глобальная блокировка (GIL)

Глобальная блокировка, или GIL, была введена, чтобы упростить работу с памятью в CPython (интерпретатор байт-кода, написан на C) и обеспечить лучшую интеграцию с C (например, расширения). GIL - это механизм блокировки, который позволяет интерпретатору Python запускать только один поток за раз. То есть только один поток может выполнять байт-код Python в любой момент времени. GIL гарантирует, что несколько потоков не будут работать параллельно.

**Характеристики GIL:**

- Одновременно может работать один поток.
- Интерпретатор Python переключается между потоками, чтобы обеспечить параллелизм.
- GIL применим только к CPython (де-факто реализация). Другие реализации, такие как Jython (Python для платформы Java) и IronPython (Python для платформы .NET), не имеют GIL.
- GIL делает однопоточные программы быстрыми.
- Для операций, связанных с вводом-выводом, GIL обычно не приносит особого вреда.
- GIL облегчает интеграцию библиотек C, не являющихся потокобезопасными. Благодаря GIL существует много высокопроизводительных расширений/модулей, написанных на C.
- Для задач, связанных с процессором, интерпретатор тактирует переключение потоков. Таким образом, один поток не блокирует другие.

Многие считают GIL слабостью, нос другой стороны благодаря ему стали возможны такие библиотеки, как NumPy, SciPy, которые заняли уникальное положение Python в научных сообществах.

## Процессы (multiprocessing)

Чтобы добиться параллелизма, в Python появился модуль multiprocessing, предоставляющий API, похожий на использование Threading. Для этого изменим предыдущий пример. Вот модифицированная версия, в которой вместо Thread используется Pool:

```
from multiprocessing.pool import Pool
import time

maxnumber = 10

def worker(number):
    sleep = maxnumber - number
    time.sleep(sleep)
    print("I am Worker {}, I slept for {} seconds".format(number, sleep))


if __name__ == '__main__':
    start_time = time.time()
    with Pool() as p:
        res = p.map_async(worker, range(maxnumber))
        print("All Processes are queued, let's see when they finish!")
        res.wait()
    print("Async is finished. Elapced:", time.time() - start_time)
```

Результат:

```
All Processes are queued, let's see when they finish!
I am Worker 9, I slept for 1 seconds
```

```
I am Worker 8, I slept for 2 seconds
I am Worker 7, I slept for 3 seconds
I am Worker 6, I slept for 4 seconds
I am Worker 5, I slept for 5 seconds
I am Worker 4, I slept for 6 seconds
I am Worker 3, I slept for 7 seconds
I am Worker 2, I slept for 8 seconds
I am Worker 1, I slept for 9 seconds
I am Worker 0, I slept for 10 seconds
Async is finished. Elapced: 10.159563302993774
```

Что изменилось? Мы импортировали модуль multiprocessing вместо threading, а вместо Thread использовали Pool. Теперь вместо многопоточности мы используем несколько процессов, которые выполняются на разных ядрах процессора. С помощью класса Pool мы также можем распределить выполнение одной функции между несколькими процессами для разных входных значений.

Время выполнения, по сравнению со Threading немного увеличилось, что можно объяснить накладными расходами на переключения в многопроцессорной системе.

## Корутины (asyncio)

Часто возникает вопрос, который задают себе многие представители сообщества Python: "Что нового в asyncio? Зачем понадобился еще один способ асинхронного ввода-вывода? Разве потоков и процессов недостаточно?

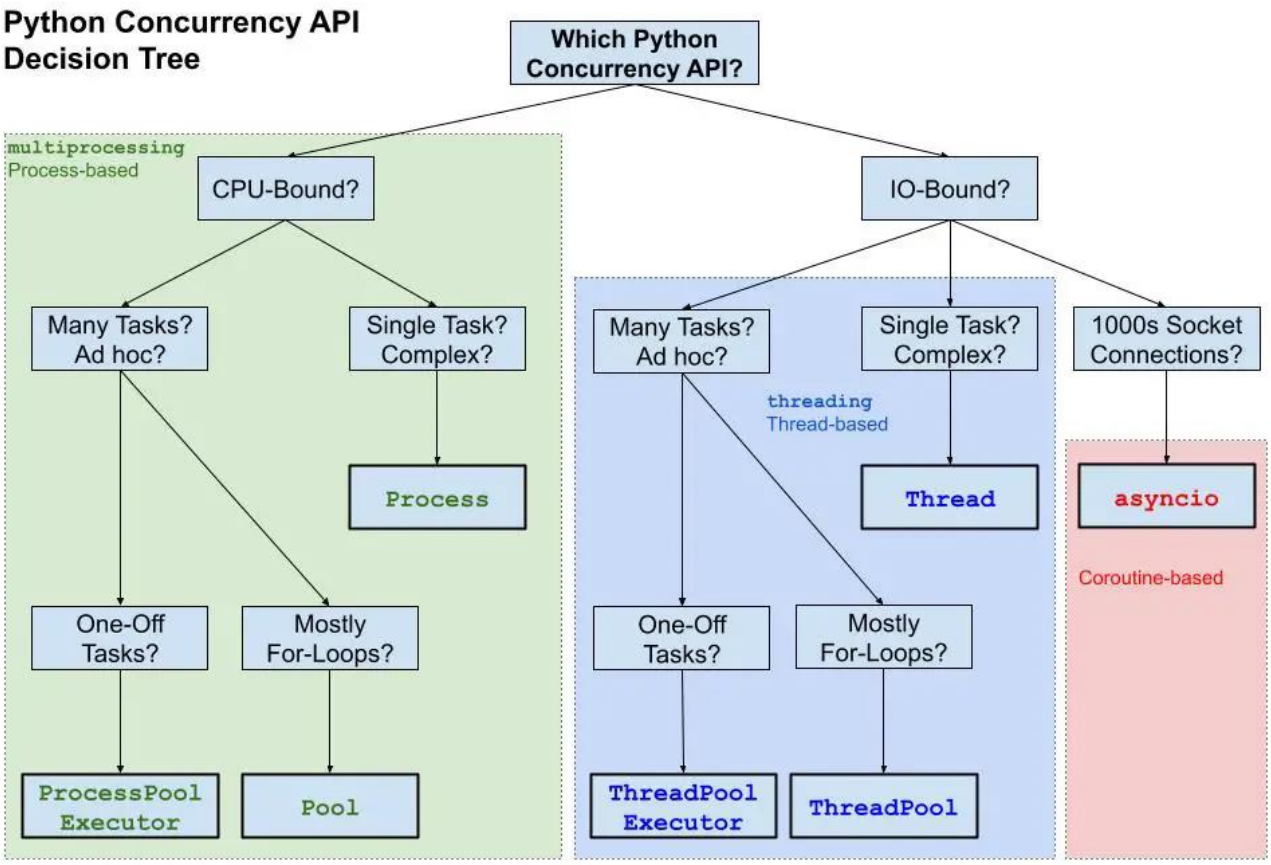
**Почему asyncio?**

Процессы порождать дорого. Поэтому для ввода-вывода в основном выбирают потоки. Известно, что ввод/вывод зависит от внешних факторов - медленные диски или неприятные сетевые задержки делают ввод/вывод часто непредсказуемым. Теперь предположим, что мы используем потоки для операций ввода-вывода. Три потока выполняют различные задачи ввода-вывода. Интерпретатору необходимо переключаться между параллельными потоками и поочередно предоставлять каждому из них некоторое время. Назовем потоки T1, T2 и T3. Все три потока начали свою операцию ввода-вывода. T3 завершает ее первым. T2 и T1 все еще ожидают ввода/вывода. Интерпретатор Python переключается на T1, но он все еще ждет. Отлично, тогда он переходит на T2, тот все еще ждет, а затем переходит на T3, который уже готов и выполняет код. Проблема в том, что T3 был готов, но интерпретатор сначала переключился между T2 и T1 — это повлекло за собой затраты на переключение, которых мы могли бы избежать, если бы интерпретатор сначала перешел на T3

**Что такое asyncio?**

Asyncio предоставляет программисту цикл событий наряду с другими полезными функциями. Событийный цикл отслеживает различные события ввода-вывода и переключается на задачи, которые уже готовы, и приостанавливает те, которые ожидают ввода-вывода. Таким образом, мы не тратим время на задачи, которые не готовы к выполнению в данный момент. Идея очень проста - есть цикл событий и у нас есть функции, которые выполняют асинхронные операции ввода-вывода. Мы передаем наши функции циклу событий и просим его выполнить их за нас. Цикл событий возвращает нам объект Future, это как обещание, что мы получим что-то в будущем. Мы храним это обещание, время от времени проверяем, есть ли у него значение, и, наконец, когда у него появляется значение, мы используем его в других операциях.

## Выбор модели асинхронного API




# Практическая часть

## Реализация для CPU-Bound модели

### Описание реализации

### Программный код

### Результаты

### Выводы

## Реализация для IO-Bound модели

### Описание реализации

### Программный код

**filewrite_classes.py**

```
import asyncio
import aiofiles
import time
import threading
from async_class import AsyncClass, AsyncObject, task, link

class AsyncTester(AsyncClass):
    async def __ainit__(self, files_count, file_size):
        self.files_count = files_count
        self.file_size = file_size

    async def run(self):
        tasks = []
```

```
        start_time = time.time()
        for i in range(self.files_count):
            tasks.append(self.writter(f'{i}.txt'))
            tasks.append(self.reader(f'{i}.txt'))
        threads_count = threading.active_count()
        await asyncio.gather(*tasks)
        return (self.file_size, (time.time() - start_time), threads_count)

    async def writter(self, filename):
        async with aiofiles.open(filename, mode='w') as file:
            content = ''.join([str(num) for num in range(self.file_size)])
            await file.write(content)

    async def reader(self, filename):
        async with aiofiles.open(filename, mode='r') as file:
            content = await file.readlines()


class ThreadTester():
    def __init__(self, files_count, file_size):
        self.files_count = files_count
        self.file_size = file_size

    def run(self):
        threads = []
        start_time = time.time()
        for i in range(self.files_count):
            thread = threading.Thread(target=self.writter, args=[i])
            thread.start()
            threads.append(thread)
            thread = threading.Thread(target=self.reader, args=[i])
            thread.start()
            threads.append(thread)
        threads_count = threading.active_count()
        for thread in threads:
            thread.join()
        return (self.file_size, (time.time() - start_time), threads_count)

    def writter(self, i):
        with open(f'{i}.txt', mode='w') as file:
            content = ''.join([str(num) for num in range(self.file_size)])
            file.write(content)

    def reader(self, i):
        with open(f'{i}.txt', mode='r') as file:
            content = file.readlines()
```

**filewrite_test.py**

```
import asyncio
import matplotlib.pyplot as plt

from filewrite_classes import AsyncTester, ThreadTester

async def async_tester(files_count, file_size):
    writter = await AsyncTester(files_count, file_size)
    result = await writter.run()
    print (
        f'Asyncio: размер файла - {result[0]}, '
        f'время выполнения - {result[1]:.2f}, '
        f'потоков - {result[2]}'
```

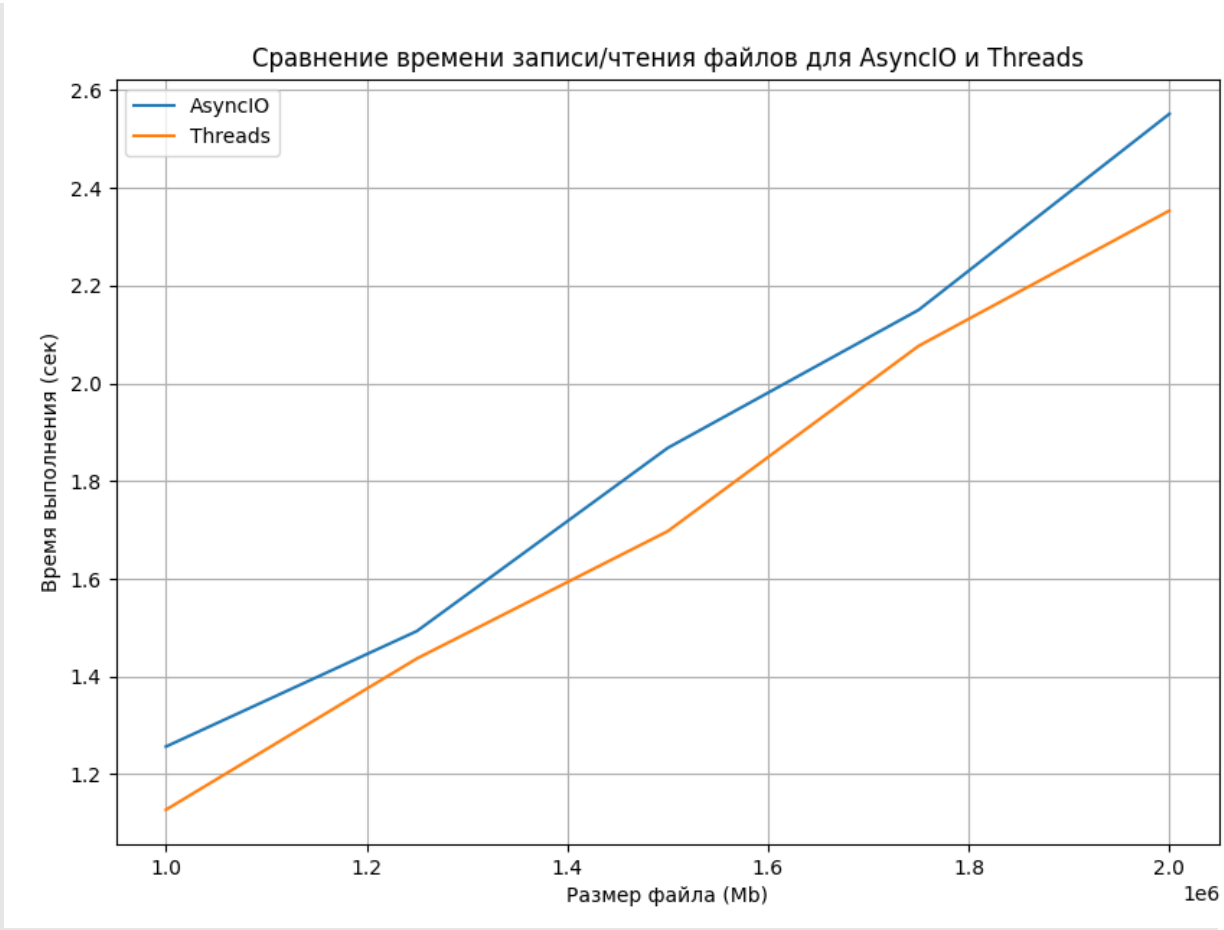
```
    )
    return result

def thread_tester(files_count, file_size):
    writter = ThreadTester(files_count, file_size)
    result = writter.run()
    print (
        f'Threads: размер файла - {result[0]}, '
        f'время выполнения - {result[1]:.2f}, '
        f'потоков - {result[2]}'
    )
    return result

files_count = 10
file_size = 2000000
results = []
xlist = []
ylist = []
for f_size in range(1000000, 2000001, 250000):
    async_result = asyncio.run(async_tester(files_count, f_size))
    thread_result = thread_tester(files_count, f_size)
    results.append(async_result)
    results.append(thread_result)
    xlist.append(f_size)
    ylist.append((async_result[1], thread_result[1]))
plt.title('Сравнение времени записи/чтения файлов для AsyncIO и Threads')
plt.xlabel('Размер файла (Mb)')
plt.ylabel('Время выполнения (сек)')
plt.plot(xlist, ylist, label = ('AsyncIO', 'Threads'))
plt.legend()
plt.grid()
plt.show()
```

## Результаты

```
Asyncio: размер файла - 1000000, время выполнения - 1.18, потоков - 1
Threads: размер файла - 1000000, время выполнения - 1.08, потоков - 8
Asyncio: размер файла - 1250000, время выполнения - 1.47, потоков - 1
Threads: размер файла - 1250000, время выполнения - 1.34, потоков - 7
Asyncio: размер файла - 1500000, время выполнения - 1.77, потоков - 1
Threads: размер файла - 1500000, время выполнения - 1.59, потоков - 4
Asyncio: размер файла - 1750000, время выполнения - 2.07, потоков - 1
Threads: размер файла - 1750000, время выполнения - 1.87, потоков - 5
Asyncio: размер файла - 2000000, время выполнения - 2.32, потоков - 1
Threads: размер файла - 2000000, время выполнения - 2.18, потоков - 6
```


Сравнение времени записи/чтения файлов для AsyncIO и Threads
AsyncIO
Threads
Время выполнения (сек)
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
1.2
1.4
1.6
1.8
2.0
1e6
Размер файла (Mb)

## Выводы