Громаков Даниил Алексеевич

МГТУ им. Н. Э. Баумана, г. Москва

АРХИТЕКТУРА КОНКУРЕНТНОГО ВЫПОЛНЕНИЯ В PYTHON: ПОТОКИ, ПРОЦЕССЫ, ASYNCIO И ИЗОЛИРОВАННЫЕ ИНТЕРПРЕТАТОРЫ

Аннотация. В работе рассматривается архитектура конкурентного выполнения в языке Python на примере актуальной реализации CPython. Актуальность темы связана с ростом числа сетевых сервисов, систем обработки данных и вычислительных приложений, для которых последовательная модель исполнения становится ограничивающим фактором. Цель исследования состоит в систематизации механизмов конкурентности Python и формировании критериев выбора между потоками, процессами, асинхронными задачами и изолированными интерпретаторами. Методика основана на анализе официальной документации Python 3.14, принятых предложений PEP и исходного кода CPython, а также на сравнении моделей по способу планирования, степени изоляции, стоимости обмена данными и возможности задействовать несколько ядер процессора. Рассмотрены глобальная блокировка интерпретатора, свободнопоточная сборка CPython, модули threading, multiprocessing и asyncio, интерфейс concurrent.futures и появившиеся в Python 3.14 средства работы с несколькими интерпретаторами. Рассмотрены практические способы организации конкурентного выполнения с использованием пулов исполнителей, средств синхронизации и передачи сообщений. Результатом является прикладная схема выбора механизма выполнения с учётом характера нагрузки, требований к изоляции и сложности синхронизации.

Ключевые слова: конкурентность, параллелизм, Python, CPython, GIL, потоки, процессы, asyncio, изолированные интерпретаторы, конкурентные паттерны.

Annotation. This paper examines the architecture of concurrent execution in Python using the current CPython implementation as the main reference. The topic is relevant to network services, data-processing systems, and computational applications in which purely sequential execution becomes a limiting factor. The aim is to systematize Python concurrency mechanisms and to formulate practical criteria for choosing between threads, processes, asynchronous tasks, and isolated interpreters. The method combines analysis of the Python 3.14 documentation, accepted Python Enhancement Proposals, and CPython source code with a comparative assessment of scheduling, isolation, communication overhead, and multi-core capability. The paper discusses the Global Interpreter Lock, the free-threaded CPython build, threading, multiprocessing, asyncio, concurrent.futures, and the multiple-interpreter facilities added to the standard library in Python 3.14. Practical approaches to concurrent execution using executor pools, synchronization mechanisms, and message passing are considered. The resulting selection framework relates the preferred execution model to workload type, isolation requirements, data-exchange costs, and synchronization complexity.

Keywords: concurrency, parallelism, Python, CPython, GIL, threads, processes, asyncio, isolated interpreters, concurrency patterns

ВВЕДЕНИЕ

Современное приложение редко выполняет только одну непрерывную последовательность вычислений. Сервер одновременно обслуживает множество соединений, программа обработки данных читает файлы и ожидает сеть, а аналитический модуль стремится распределить независимые расчёты между вычислительными ядрами. Поэтому конкурентность следует рассматривать не как отдельный приём оптимизации, а как один из способов структурирования программной системы.

В инженерной практике понятия конкурентности и параллелизма часто смешиваются. Конкурентная система допускает продвижение нескольких задач в одном временном интервале, даже если процессор фактически

переключается между ними. Параллельная система исполняет несколько фрагментов работы одновременно на разных вычислительных ресурсах. Конкурентность прежде всего описывает организацию программы, тогда как параллелизм характеризует физическое исполнение. Ограничение ускорения долей программы, которые нельзя распараллелить, традиционно связывают с законом Амдала [1-2].

Python предоставляет несколько моделей, которые внешне решают похожую задачу, но обладают принципиально разными свойствами. Потоки разделяют память процесса; процессы изолируют адресные пространства; asyncio организует кооперативное переключение корутин внутри цикла событий; изолированные интерпретаторы занимают промежуточное положение между потоками и процессами. Дополнительную сложность создаёт то, что термин Python обозначает язык, тогда как особенности глобальной блокировки относятся прежде всего к реализации CPython.

В Python 3.13 была представлена свободнопоточная сборка CPython, позволяющая отключить GIL, а в Python 3.14 она перешла в статус официально поддерживаемой, но по-прежнему необязательной конфигурации [8-9, 13]. Одновременно Python 3.14 включил в стандартную библиотеку средства управления изолированными интерпретаторами и новый InterpreterPoolExecutor [7, 11-12]. Эти изменения делают прежнее упрощённое правило «для ввода-вывода — потоки, для вычислений — процессы» недостаточным.

Целью работы является анализ архитектуры конкурентного выполнения в Python и формирование практических рекомендаций по выбору модели исполнения. Объектом исследования выступают средства стандартной библиотеки и runtime CPython, а предметом — механизмы планирования, изоляции, передачи данных и синхронизации конкурентных задач.

Для достижения цели решаются следующие задачи:

1. разграничить конкурентность и параллелизм применительно к Python-программам;
2. описать роль процесса, потока, состояния интерпретатора и асинхронной задачи;
3. проанализировать влияние GIL и свободнопоточного режима на многопоточное выполнение;
4. сопоставить threading, multiprocessing, asyncio и несколько интерпретаторов;
5. рассмотреть средства синхронизации и типовые ошибки;
6. сформулировать практические критерии выбора модели конкурентного выполнения.

МАТЕРИАЛЫ И МЕТОДЫ

Исследование выполнено как технический обзор. Основными материалами служили официальная документация Python 3.14, тексты принятых PEP 684, PEP 703, PEP 734 и PEP 779, а также исходный код ветки CPython 3.14 [3-13, 17]. Для общесистемных понятий использовались классические работы по операционным системам, параллельным вычислениям и взаимодействующим последовательным процессам [1-2, 16].

Сравнение моделей проводилось по пяти признакам: единица планирования; способ разделения памяти; возможность физического исполнения Python-кода на нескольких ядрах; стоимость запуска и обмена данными; типовые риски корректности. Приведённые листинги являются самостоятельными примерами автора и предназначены для демонстрации структуры решений, а не для получения универсальных численных оценок производительности.

УРОВНИ ВЫПОЛНЕНИЯ: ПРОЦЕСС, ИНТЕРПРЕТАТОР, ПОТОК И ЗАДАЧА

Операционная система планирует потоки, принадлежащие процессам. Процесс объединяет адресное пространство, дескрипторы и другие ресурсы, а поток задаёт последовательность исполняемых инструкций. Потоки одного

процесса видят общую память, поэтому обмен данными между ними дешёв, но любой изменяемый объект может стать источником состояния гонки [1].

Внутри процесса CPython существует состояние интерпретатора PyInterpreterState. Оно включает таблицу модулей, встроенные объекты и значительную часть состояния runtime. С каждым активным потоком связывается PyThreadState — контекст исполнения конкретного потока в конкретном интерпретаторе [10-12]. Один процесс может содержать несколько интерпретаторов, а один интерпретатор — обслуживаться несколькими потоками.

Асинхронная Task не является потоком операционной системы. Это объект уровня библиотеки asyncio, который планирует продвижение корутины. Несколько Task обычно выполняются одним потоком: текущая корутина добровольно отдаёт управление в выражении await, после чего цикл событий выбирает другую готовую работу [5-6].

Таким образом, единицы выполнения образуют вложенную структуру. Процесс задаёт границу адресного пространства; интерпретатор задаёт контекст Python-runtime; поток является единицей планирования ОС; корутина и Task являются единицами кооперативного планирования внутри asyncio. Обобщённая схема уровней организации конкурентного выполнения представлена на рисунке 1.

РИСУНОК 1 – УРОВНИ ОРГАНИЗАЦИИ КОНКУРЕНТНОГО ВЫПОЛНЕНИЯ В CPYTHON

FIGURE 1 – LAYERS OF CONCURRENT EXECUTION IN CPYTHON

КОНКУРЕНТНОСТЬ И ПАРАЛЛЕЛИЗМ ДЛЯ ЗАДАЧ РАЗНЫХ ТИПОВ

Практический выбор механизма начинается с определения преобладающего ожидания. I/O-bound задача большую часть времени ожидает сеть, диск, базу данных или внешний сервис. CPU-bound задача расходует время на вычисления в пользовательском коде. Смешанная нагрузка включает обе составляющие и часто требует комбинации нескольких моделей.

Для I/O-bound работы ускорение достигается не за счёт более быстрого выполнения отдельной операции, а за счёт перекрытия периодов ожидания. Пока одна задача не может продолжаться, ресурс используется другой задачей. Для CPU-bound работы необходимы независимые вычислительные ресурсы: процессы, изолированные интерпретаторы или свободнопоточная сборка при корректно синхронизированном коде.

Количество конкурентных задач не следует механически приравнивать к количеству ядер. Сетевой клиент может поддерживать сотни ожидающих операций в одном потоке asyncio, тогда как число одновременно работающих CPU-задач обычно ограничивают доступными ядрами и пропускной

способностью памяти. Избыточный параллелизм приводит к переключениям контекста, росту очередей, конкуренции за кэш и памяти.

ГЛОБАЛЬНАЯ БЛОКИРОВКА ИНТЕРПРЕТАТОРА

В стандартной сборке CPython GIL допускает исполнение Python-байткода только одним потоком данного интерпретатора в конкретный момент времени [3, 8]. Блокировка защищает значительную часть внутреннего состояния runtime, включая операции, связанные с управлением объектами. Она не запрещает существование множества потоков и не означает, что программа всегда использует только одно ядро: расширения на C могут освобождать GIL, а операции ввода-вывода обычно переводят поток в ожидание.

Следствие GIL состоит в том, что два потока одного интерпретатора в обычной сборке не обеспечивают линейного ускорения чистого Python-кода, интенсивно использующего процессор. При этом потоки остаются эффективными для задач с ожиданием, поскольку блокированный поток не удерживает возможность исполнения полезной работы другим потоком [3].

GIL нельзя считать заменой пользовательской синхронизации. Логическая операция высокого уровня может состоять из нескольких байткод-инструкций, между которыми произойдёт переключение. Кроме того, код должен сохранять корректность в свободнопоточной сборке и при вызовах расширений, освобождающих GIL. Поэтому доступ к общему изменяемому состоянию необходимо явно защищать Lock, RLock, Condition либо заменять передачей сообщений.

PEP 703 определил конфигурацию CPython без GIL и потребовал изменений подсчёта ссылок, управления памятью и защиты контейнеров [8]. В Python 3.14 свободнопоточная сборка официально поддерживается, но не является конфигурацией по умолчанию [9, 13]. Совместимость сторонних C-расширений остаётся существенным ограничением: неподготовленный

модуль может потребовать повторного включения GIL. Характер чередования потоков при наличии GIL показан на рисунке 2.

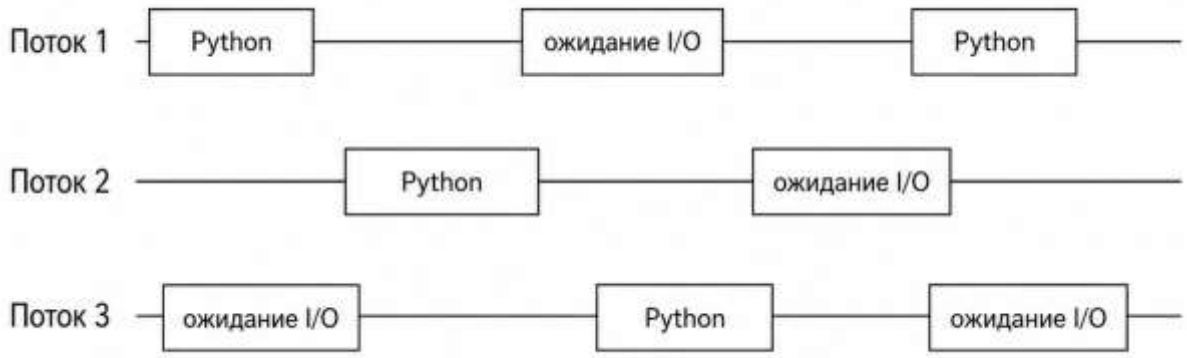

РИСУНОК 2 – ЧЕРЕДОВАНИЕ ПОТОКОВ В ИНТЕРПРЕТАТОРЕ С GIL
FIGURE 2 – THREAD INTERLEAVING IN A GIL-ENABLED INTERPRETER

ПОТОКОВАЯ МОДЕЛЬ THREADING

Модуль threading предоставляет объект Thread и набор примитивов синхронизации поверх потоков операционной системы [3]. Все потоки одного интерпретатора разделяют объекты Python, файловые дескрипторы и другие ресурсы процесса. Благодаря этому функция может получить ссылку на уже существующую структуру данных без сериализации.

Низкая стоимость обмена является одновременно преимуществом и источником сложности. Если два потока изменяют общий объект без согласованного протокола, результат зависит от относительного порядка операций. Базовым средством исключения является Lock. RLock допускает повторный захват владельцем, Condition связывает ожидание условия с блокировкой, Event распространяет сигнал, Semaphore ограничивает число одновременных участников, а Barrier синхронизирует достижение общей точки [3].

В прикладном коде предпочтительнее управлять группой однотипных задач через ThreadPoolExecutor. Пул повторно использует ограниченное число потоков, возвращает Future и отделяет постановку задачи от ожидания результата [7]. Такой подход упрощает обработку исключений и завершение

ресурсов, однако не устраняет взаимоблокировки: задача пула не должна без необходимости ждать результат другой задачи того же исчерпанного пула.

Потоки рациональны для блокирующего I/O, интеграции с синхронными библиотеками, фоновых служебных операций и вычислений в расширениях, которые освобождают GIL. Для чистого CPU-bound Python-кода в обычной сборке следует рассматривать процессы либо интерпретаторы.

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from time import sleep
def load_fragment(fragment_id: int) -> tuple[int, int]:
    # sleep моделирует ожидание внешнего источника данных.
    sleep(0.05 + 0.01 * (fragment_id % 3))
    return fragment_id, fragment_id * 10
def collect_fragments(ids: list[int]) -> dict[int, int]:
    result: dict[int, int] = {}
    with ThreadPoolExecutor(max_workers=4,
                            thread_name_prefix="loader") as executor:
        futures = [executor.submit(load_fragment, value) for value in ids]
        for future in as_completed(futures):
            key, payload = future.result()
            result[key] = payload
    return result
if __name__ == "__main__":
    print(collect_fragments(list(range(8))))
```

ЛИСТИНГ 1 – ВЫПОЛНЕНИЕ I/O-ПОДОБНЫХ ОПЕРАЦИЙ В THREADPOOLEXECUTOR

LISTING 1 – EXECUTING I/O-LIKE OPERATIONS WITH THREADPOOLEXECUTOR

В листинге 1 число одновременно исполняемых операций ограничено четырьмя потоками. Контекстный менеджер гарантирует корректное

завершение пула, а as_completed позволяет обрабатывать результаты по мере готовности, не навязывая порядок исходного списка.

ПРОЦЕССНАЯ МОДЕЛЬ MULTIPROCESSING

Модуль multiprocessing создаёт отдельные процессы и тем самым обходит ограничение GIL обычной сборки: у каждого дочернего процесса есть собственный интерпретатор и собственная память [4]. Модель подходит для независимых вычислений, которые можно представить в виде функций с сериализуемыми аргументами и результатами.

Изоляция повышает устойчивость архитектуры: случайное изменение объекта в одном процессе не затрагивает память другого. Цена изоляции — запуск интерпретатора, сериализация через pickle, копирование данных и межпроцессное взаимодействие. Передача большого массива при каждой задаче способна поглотить выигрыш от параллельного вычисления, поэтому крупные неизменяемые данные целесообразно инициализировать один раз в рабочем процессе либо размещать в разделяемой памяти.

В Python 3.14 метод fork больше не используется по умолчанию ни на одной платформе. На POSIX основным стал forkserver, а Windows и macOS используют spawn; код, требующий fork, должен выбирать его явно [4]. Из этого следует практическое требование: точка входа должна быть защищена конструкцией if __name__ == "__main__", а передаваемые функции должны быть доступны для импорта.

Высокоуровневый ProcessPoolExecutor предоставляет тот же интерфейс Executor, что и потоковый пул [7]. Это облегчает замену модели, но не делает стоимость передачи данных одинаковой. До распараллеливания необходимо оценить гранулярность: задача должна выполнять достаточно полезной работы, чтобы компенсировать накладные расходы процесса.

```
from concurrent.futures import ProcessPoolExecutor
from math import isqrt
def count_primes(limit: int) -> int:
```

```
        count = 0
        for value in range(2, limit + 1):
            prime = True
            for divisor in range(2, isqrt(value) + 1):
                if value % divisor == 0:
                    prime = False
                    break
            count += int(prime)
        return count
    def main() -> None:
        limits = [35_000, 37_000, 39_000, 41_000]
        with ProcessPoolExecutor(max_workers=4) as executor:
            results = list(executor.map(count_primes, limits))
        print(dict(zip(limits, results)))
    if __name__ == "__main__":
        main()
```

ЛИСТИНГ 2 – РАСПРЕДЕЛЕНИЕ CPU-BOUND ЗАДАЧ МЕЖДУ ПРОЦЕССАМИ

LISTING 2 – DISTRIBUTING CPU-BOUND TASKS ACROSS PROCESSES

Листинг 2 демонстрирует независимые вычисления с небольшими аргументами и результатами. Такая форма благоприятна для процесса: стоимость сериализации мала по сравнению с объёмом расчёта. Для коротких функций или больших передаваемых объектов результат может оказаться обратным.

АСИНХРОННАЯ МОДЕЛЬ ASYNCIO

asyncio организует конкурентность в основном внутри одного потока. Цикл событий хранит готовые callback и Task, ожидает события ввода-вывода и последовательно исполняет очередной участок корутины. Когда корутина

достигает await и ожидаемый объект ещё не готов, её состояние сохраняется, а управление возвращается циклу [5-6].

Кооперативная модель уменьшает число переключений контекста ОС и позволяет обслуживать большое количество соединений без создания потока на каждое из них. Однако справедливость зависит от поведения кода: длительная синхронная функция блокирует не только текущую корутину, но и весь цикл событий. Официальная документация рекомендует выносить блокирующую работу через asyncio.to_thread или loop.run_in_executor, используя потоковый, процессный либо интерпретаторный пул [6-7].

Основными awaitable-объектами являются coroutine, Task и Future [5]. Task связывает корутину с планировщиком, а Future представляет результат, который будет получен позднее. TaskGroup реализует структурированную конкурентность: дочерние задачи создаются внутри ограниченной области, а выход из контекстного менеджера означает ожидание их завершения и согласованную обработку ошибок.

Асинхронные Lock, Event, Condition, Semaphore и Queue похожи по назначению на потоковые аналоги, но не являются thread-safe и предназначены только для задач одного событийного контекста [15]. Наличие одного потока не устраняет логические гонки: между чтением и записью общего состояния может находиться await, допускающий выполнение другой задачи. Упрощённая схема работы цикла событий представлена на рисунке 3.

РИСУНОК 3 – УПРОЩЁННАЯ СХЕМА ЦИКЛА СОБЫТИЙ ASYNCIO

FIGURE 3 – SIMPLIFIED ASYNCIO EVENT-LOOP CYCLE

ИЗОЛИРОВАННЫЕ ИНТЕРПРЕТАТОРЫ И INTERPRETERPOOLEXECUTOR

PEP 684 перенёс GIL на уровень интерпретатора при достаточной изоляции runtime-состояния [10]. Благодаря этому два потока, работающие в разных интерпретаторах одного процесса, могут исполнять Python-код параллельно на разных ядрах, не разделяя одну глобальную блокировку.

Python 3.14 добавил модуль concurrent.interpreters и InterpreterPoolExecutor [7, 11-12]. Каждый рабочий поток интерпретаторного пула использует собственный интерпретатор, отдельные модули и глобальные переменные. Изменяемый объект нельзя просто передать по ссылке; аргументы и результаты обычно сериализуются, а взаимодействие строится через специально поддерживаемые очереди или другие явные каналы.

По сравнению с процессами интерпретаторы сохраняют единый процесс и потенциально требуют меньше системных ресурсов. По сравнению с обычными потоками они уменьшают объём неявно разделяемого состояния и

обеспечивают многоядерное исполнение даже в сборке с GIL. Ограничениями являются зрелость экосистемы, совместимость расширений и необходимость отдельно импортировать модули в каждом интерпретаторе [12].

Эта модель особенно интересна для независимых CPU-задач среднего размера, когда изоляция полезна, но создание процессов слишком дорого. Тем не менее выбор должен подтверждаться измерениями на конкретной нагрузке: сериализация и повторная инициализация модулей могут изменить итоговый баланс.

```
from concurrent.futures import InterpreterPoolExecutor
from hashlib import sha256
def hash_block(seed: int) -> str:
    data = str(seed).encode()
    for _ in range(120_000):
        data = sha256(data).digest()
    return data.hex()
def main() -> None:
    with InterpreterPoolExecutor(max_workers=4) as executor:
        hashes = list(executor.map(hash_block, range(4)))
    for index, digest in enumerate(hashes):
        print(index, digest[:16])
if __name__ == "__main__":
    main()
```

ЛИСТИНГ 3 – CPU-BOUND ЗАДАЧИ В INTERPRETERPOOLEXECUTOR PYTHON 3.14

LISTING 3 – CPU-BOUND TASKS WITH PYTHON 3.14 INTERPRETERPOOLEXECUTOR

Листинг 3 требует Python 3.14. Функция и аргументы сериализуются для рабочего интерпретатора, поэтому они должны быть пригодны для передачи.

Независимые значения seed не требуют общего изменяемого состояния и хорошо соответствуют модели изолированных исполнителей.

СВОБОДНОПОТОЧНАЯ СБОРКА CPYTHON

Свободнопоточная сборка создаётся с поддержкой отключённого GIL и позволяет нескольким потокам одного интерпретатора одновременно выполнять Python-код [8, 13]. Проверить конфигурацию можно через sysconfig, а фактическое состояние GIL — средствами sys. В Python 3.14 режим официально поддерживается, однако пользователь должен отдельно установить соответствующую сборку или собрать интерпретатор.

Отсутствие GIL не превращает существующий многопоточный код автоматически в масштабируемый. Общие структуры требуют блокировок, а слишком частые синхронизации сохраняют последовательный участок. Внутренние блокировки list и dict защищают реализацию от повреждения, но документация не рекомендует использовать их как контракт логической атомарности [13].

Свободнопоточный режим имеет дополнительные накладные расходы на однопоточное выполнение и память. Кроме того, расширение, не заявившее поддержку, может включить GIL при импорте [13]. Поэтому миграция включает проверку зависимостей, поиск скрытых гонок и повторное профилирование. Наиболее перспективны приложения, уже использующие потоки и имеющие крупные независимые вычислительные участки.

СИНХРОНИЗАЦИЯ И ПЕРЕДАЧА СООБЩЕНИЙ

Конкурентная программа нуждается не только в запуске задач, но и в правилах владения данными. Разделяемое изменяемое состояние требует критических секций. Передача сообщений уменьшает число мест, где несколько исполнителей могут изменить один объект. Идея взаимодействующих последовательных процессов подчёркивает, что независимые компоненты проще согласовывать через явные каналы [16].

Для потоков стандартный модуль queue реализует многопроизводительную и многопотребительскую очередь с необходимыми блокировками [14]. Ограниченный maxsize полезен как механизм обратного давления: производитель не накапливает неограниченный объём работы. Методы task_done и join позволяют отделить факт извлечения элемента от завершения его обработки.

Процессы используют Queue и Pipe модуля multiprocessing, а также менеджеры и shared_memory. Передача через очередь сериализует объекты, тогда как разделяемая память требует отдельного протокола синхронизации. Для asyncio применяются asyncio.Queue и асинхронные примитивы; они не должны использоваться для связи между потоками [15].

При выборе блокировки важно минимизировать область критической секции, не выполнять внутри неё неопределённо долгие операции и соблюдать единый порядок захвата нескольких блокировок. Тайм-ауты и отмена должны рассматриваться как часть протокола, а не как позднее дополнение.

РЕЗУЛЬТАТЫ

Анализ показывает, что единой наилучшей модели не существует. Потоки минимизируют стоимость доступа к общим данным, но увеличивают риск гонок. Процессы обеспечивают наиболее понятную изоляцию, но требуют дорогого обмена. asyncio эффективно масштабирует ожидание I/O, однако чувствителен к любой блокирующей функции. Интерпретаторы добавляют вариант многоядерного выполнения в одном процессе, а свободнопоточная сборка переносит ответственность за корректную синхронизацию на весь стек программы.

Для практического выбора сформулирован следующий порядок: сначала исключить ненужную конкурентность; затем определить CPU- или I/O-характер нагрузки; оценить объём разделяемого состояния; выбрать минимально сложную модель; ограничить число одновременных операций;

добавить отмену и завершение; после этого измерить пропускную способность, задержку и потребление памяти.

ЗАКЛЮЧЕНИЕ

В работе систематизированы основные механизмы конкурентного выполнения Python применительно к CPython 3.14. Показано, что процесс, интерпретатор, поток и асинхронная Task образуют разные уровни планирования и не являются взаимозаменяемыми понятиями.

Стандартная сборка CPython сохраняет GIL на уровне интерпретатора, поэтому потоки одного интерпретатора прежде всего полезны для перекрытия ожидания и работы с общими данными. multiprocessing обеспечивает многоядерное выполнение посредством отдельных процессов и оплачивает изоляцию запуском и сериализацией. asyncio реализует кооперативную конкурентность с низкой стоимостью Task и особенно эффективно для неблокирующего I/O.

Изолированные интерпретаторы, доступные через стандартную библиотеку Python 3.14, предоставляют самостоятельный runtime-контекст в одном процессе и отдельный GIL для каждого интерпретатора. InterpreterPoolExecutor формирует новую практическую альтернативу процессному пулу. Свободнопоточная сборка снимает обязательный GIL для потоков одного интерпретатора, но требует зрелой поддержки зависимостей и дисциплины синхронизации.

Выбор модели должен определяться характером нагрузки, объёмом разделяемого состояния, требованиями к изоляции и стоимостью обмена данными. Наиболее устойчивые решения используют ограниченную конкурентность, явный протокол завершения, передачу сообщений и структурированное управление временем жизни задач. Окончательное решение необходимо подтверждать профилированием на целевой платформе.

Литература

1. Silberschatz A., Galvin P. B., Gagne G. Operating System Concepts. 10th ed. Hoboken, NJ: Wiley; 2018. 976 p.
2. Amdahl G. M. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Conference Proceedings. 1967;30:483–485. DOI: 10.1145/1465482.1465560.
3. Python Software Foundation. threading — Thread-based parallelism. Python 3.14 documentation. URL: https://docs.python.org/3.14/library/threading.html (дата обращения: 18.06.2026).
4. Python Software Foundation. multiprocessing — Process-based parallelism. Python 3.14 documentation. URL: https://docs.python.org/3.14/library/multiprocessing.html (дата обращения: 11.06.2026).
5. Python Software Foundation. Coroutines and Tasks. Python 3.14 documentation. URL: https://docs.python.org/3.14/library/asyncio-task.html (дата обращения: 11.06.2026).
6. Python Software Foundation. Developing with asyncio. Python 3.14 documentation. URL: https://docs.python.org/3.14/library/asyncio-dev.html (дата обращения: 11.06.2026).
7. Python Software Foundation. concurrent.futures — Launching parallel tasks. Python 3.14 documentation. URL: https://docs.python.org/3.14/library/concurrent.futures.html (дата обращения: 20.06.2026).
8. Gross S. PEP 703 — Making the Global Interpreter Lock Optional in CPython. URL: https://peps.python.org/pep-0703/ (дата обращения: 19.06.2026).
9. Wouters T., Page M., Gross S. PEP 779 — Criteria for supported status for free-threaded Python. URL: https://peps.python.org/pep-0779/ (дата обращения: 03.07.2026).

10. Snow E. PEP 684 — A Per-Interpreter GIL. URL: https://peps.python.org/pep-0684/ (дата обращения: 27.06.2026).
11. Snow E. PEP 734 — Multiple Interpreters in the Stdlib. URL: https://peps.python.org/pep-0734/ (дата обращения: 27.06.2026).
12. Python Software Foundation. concurrent.interpreters — Multiple interpreters in the same process. Python 3.14 documentation. URL: https://docs.python.org/3.14/library/concurrent.interpreters.html (дата обращения: 01.07.2026).
13. Python Software Foundation. Python support for free threading. Python 3.14 documentation. URL: https://docs.python.org/3.14/howto/free-threading-python.html (дата обращения: 01.07.2026).
14. Python Software Foundation. queue — A synchronized queue class. Python 3.14 documentation. URL: https://docs.python.org/3.14/library/queue.html (дата обращения: 23.06.2026).
15. Python Software Foundation. Synchronization Primitives. Python 3.14 documentation. URL: https://docs.python.org/3.14/library/asyncio-sync.html (дата обращения: 23.06.2026).
16. Hoare C. A. R. Communicating Sequential Processes. Communications of the ACM. 1978;21(8):666–677. DOI: 10.1145/359576.359585.
17. Python Software Foundation. CPython source code, branch 3.14. URL: https://github.com/python/cpython/tree/3.14 (дата обращения: 27.06.2026).