УДК 004.42
ББК 32.372
Ф28

Фаулер М.
Ф28 Asyncio и конкурентное программирование на Python / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2022. – 398 с.: ил.

ISBN 978-5-93700-166-5

Эта книга адресована разработчикам средней и высокой квалификации, которые хотят использовать средства конкурентности в Python для повышения производительности, пропускной способности и отзывчивости приложений.

Из начальных глав читатель узнает, как работает asyncio, как написать первое реальное приложение и как использовать базовые функции asyncio API для конкурентного выполнения сопрограмм. Затем речь пойдет о практическом применении конкурентности – например, о том, как отправить несколько конкурентных веб-запросов или запросов к базе данных, как управлять потоками и процессами, строить веб-приложения и решать вопросы синхронизации. Рассматривается широкий круг применений от API на основе веба до командных приложений, так что книга будет полезной в решении многих реальных задач.

УДК 004.42
ББК 32.372

ISBN 978-1-6172-9866-0 (англ.)
ISBN 978-5-93700-166-5 (рус.)

Оглавление

Содержание

Предисловие

Почти 20 лет назад я начал профессиональную карьеру в области программной инженерии, написав приложение для управления масс-спектрометрами и другими лабораторными приборами и анализа поступающих от них данных. В приложении использовались языки Matlab, C++ и VB.net. Меня всегда охватывал восторг при виде того, как строчка кода заставляет машину двигаться по моему желанию, и с тех пор я понял, что хочу заниматься только программной инженерией. Шли годы, мои интересы постепенно сместились в сторону разработки API и распределенных систем, в основном на Java и Scala, но попутно я активно изучал Python.

Я впервые столкнулся с Python примерно в 2015 году, работа была связана главным образом с построением конвейера машинного обучения, который принимал данные от датчиков и на их основе делал предсказания относительно действий носителя датчика – например, отслеживание сна, подсчет числа шагов, переходы из положения сидя в положение стоя и т. д. Тогда этот конвейер был медленным настолько, что это стало проблемой для клиентов. Одним из способов, которые я применил для решения проблемы, стало использование конкурентности. Копаясь в доступной информации о конкурентном программировании на Python, я обнаружил, что разобраться в ней труднее, чем в привычном мне мире Java. Почему многопоточность работает не так, как в Java? Есть ли смысл использовать многопроцессную обработку? Что такое глобальная блокировка интерпретатора и зачем она нужна? На тему конкурентности в Python книг было немного, а знания в основном были разбросаны по документации и блогам разного качества. И сегодня ситуация мало чем отличается. Ресурсов стало больше, но ландшафт по-прежнему скудный, разъединенный и не такой дружелюбный к начинающим, каким должен, по идее, быть.

Разумеется, за последние несколько лет многое изменилось. Тогда пакет asyncio пребывал в младенчестве, а теперь стал важным модулем в Python. Ныне модели однопоточной конкурентности и сопрограммы являются одним из базовых компонентов Python в дополнение к многопоточности и многопроцессности. Это значит, что ландшафт конкурентности в Python стал шире и сложнее, но так и не обрел исчерпывающих ресурсов, к которым мог бы обратиться желающий изучить его.

Я написал эту книгу, чтобы заполнить этот пробел, конкретно в области asyncio и однопоточной конкурентности. Я хотел сделать сложную и плохо документированную тему однопоточной конкурентности более доступной разработчикам всех уровней. Кроме того, я хотел написать книгу, благодаря которой читатель стал бы лучше понимать общие проблемы конкурентности, выходящие за рамки Python. В таких каркасах, как Node.js, и таких языках, как Kotlin, имеются модели однопоточной конкурентности и сопрограммы, поэтому приведенные здесь сведения будут полезны и при работе с этими инструментами. Надеюсь, что книга окажется полезной читателям-разработчикам в повседневной работе, – и не только в Python, но и вообще в области конкурентного программирования.

Об этой книге

Эта книга написана для тех, кто хочет научиться использовать средства конкурентности в Python, чтобы повысить производительность, пропускную способность и отзывчивость приложений. Сначала мы рассмотрим базовые вопросы конкурентности, объясним, как работает модель однопоточной конкурентности в asyncio, а также расскажем о принципах работы сопрограмм и синтаксисе async/await. Затем перейдем к практическим приложениям конкурентности, например: как отправить несколько конкурентных веб-запросов или запросов к базе данных, как управлять потоками и процессами, строить веб-приложения и решать вопросы синхронизации.

Кому стоит прочитать эту книгу?

Книга адресована разработчикам средней и высокой квалификации, которые хотят разобраться в конкурентности и использовать ее в уже написанных или новых приложениях. Одна из целей книги – объяснить сложные вопросы простым и понятным языком. Предварительного знакомства с конкурентностью не требуется, хотя оно, конечно, не помешает. Мы рассмотрим широкий круг применений: от API на основе веба до командных приложений, так что книга будет полезна для решения многих задач, с которыми сталкиваются разработчики.

Структура книги

В этой книге 14 глав, в которых рассматриваются постепенно усложняющиеся темы. Последующие главы основаны на материале предыдущих.

- **Глава 1** посвящена базовым вопросам конкурентности в Python. Мы узнаем о задачах, ограниченных быстродействием процессора (счетных) и производительностью ввода-вывода, а также

начнем рассказ о том, как работает модель однопоточной конкурентности.

- **Глава 2** посвящена основам сопрограмм asyncio и использованию синтаксиса async/await в конкурентных приложениях.
- В **главе 3** рассматриваются неблокирующие сокеты и селекторы, а также описывается построение эхо-сервера с применением asyncio.
- **Глава 4** посвящена отправке нескольк конкурентных веб-запросов. Попутно мы больше узнаем о том, как использовать asyncio API для конкурентного выполнения сопрограмм.
- Тема **главы 5** – конкурентное выполнение запросов к базе данных с использованием пула подключений. Также мы узнаем об асинхронных контекстных менеджерах и асинхронных генераторах в контексте баз данных.
- В **главе 6** обсуждается многопроцессная обработка, в частности использование asyncio для выполнения счетных задач. Для демонстрации будет построено сообщение типа map-reduce.
- **Глава 7** посвящена многопоточной обработке, а особенно ее использованию в сочетании с asyncio для обработки блокирующего ввода-вывода. Это полезно при работе с библиотеками, в которые поддержка asyncio не встроена, что не мешает им получать выгоды от конкурентного выполнения.
- **Глава 8** посвящена потоковой обработке и протоколам. Мы напишем сервер и клиент чата, способные конкурентно обрабатывать несколько пользователей.
- В **главе 9** рассматриваются веб-приложения на основе asyncio и асинхронный интерфейс серверного шлюза, ASGI. Мы изучим несколько каркасов, поддерживающих ASGI, и обсудим, как строить в них веб-API. Также поговорим о технологии WebSockets.
- В **главе 10** описывается, как с помощью веб-API на основе asyncio построить гипотетическую микросервисную архитектуру.
- **Глава 11** посвящена проблемам синхронизации в модели однопоточной конкурентности и методам их решения. Мы рассмотрим блокировки, семафоры, события и условия.
- **Глава 12** посвящена асинхронным очередям. С их помощью мы построим веб-приложение, мгновенно отвечающее на клиентские запросы, хотя в фоновом режиме производятся длительные операции.
- В **главе 13** обсуждается создание подпроцессов и управление ими. Мы покажем, как читать и передавать данные подпроцессу.
- В **главе 14** рассматриваются дополнительные темы, в том числе принудительный переход к новой итерации цикла событий, контекстные переменные и создание собственного цикла событий. Эта информация полезна прежде всего проектировщикам asyncio API и читателям, интересующимся внутренним устройством цикла событий в asyncio.

Как минимум следует прочитать первые четыре главы, чтобы понять, как работает asyncio, как написать свое первое реальное приложение и как использовать базовые функции asyncio API для конкурентного выполнения сопрограмм (рассматриваются в главе 4). После этого вы вольны читать книгу в любом порядке, отвечающем вашим интересам.

О примерах кода

В этой книге много примеров кода как в пронумерованных листингах, так и в виде небольших фрагментов. Некоторые листинги используются повторно путем импорта в той же главе, а иногда и в нескольких главах. Предполагается, что код, используемый в нескольких главах, помещен в модуль `util`; мы создадим его в главе 2. Предполагается также, что для каждого отдельного листинга будет создан модуль с именем `chapter_{номер_главы}`, а код будет помещен в файл с именем `listing_{номер_главы}_{номер_листинга}.py`, принадлежащий этому модулю. Например, код из листинга 2.2 в главе 2 должен находиться в файле `listing_2_2.py`, принадлежащем модулю `chapter_2`.

В нескольких местах приводятся сведения о производительности, например время работы программы или количество веб-запросов в секунду. Примеры кода выполнялись на компьютере 2019 MacBook Pro с 8-ядерным процессором Intel Core i9 с тактовой частотой 2,4 ГГц и 32 Гб памяти DDR4, работающей на частоте 2667 МГц; использовалось также гигабитное беспроводное подключение к интернету. На вашей машине цифры могут получиться другими, как и коэффициенты ускорения или иного улучшения производительности.

Исполняемые фрагменты кода можно найти в онлайновой версии книги на сайте https://livebook.manning.com/book/python-concurrency-with-asyncio. Полный исходный код можно скачать бесплатно с сайта издательства Manning по адресу https://www.manning.com/books/python-concurrency-with-asyncio, также он доступен на Github по адресу https://github.com/concurrency-in-python-with-asyncio.

Форум на сайте liveBook

Приобретение этой книги открывает бесплатный доступ к платформе liveBook онлайнового чтения, созданной издательством Manning. Средства обсуждения на liveBook позволяют присоединять комментарии как к книге в целом, так и к отдельным разделам или абзацам. Совсем несложно добавить примечания для себя, задать или ответить на технический вопрос и получить помощь от автора и других пользователей. Для доступа к форуму перейдите по адресу https://livebook.manning.com/#!/book/python-concurrency-with-asyncio/discussion. Узнать о форумах Manning и правилах поведения на них можно по адресу https://livebook.manning.com/#!/discussion.

Издательство Manning обязуется предоставлять площадку для содержательного диалога между читателями, а также между читателями

и автором. Но это обязательство не подразумевает какого-то конкретного количества часов присутствия автора, участие которого в работе форума остается добровольным (и не оплачивается). Мы рекомендуем задавать автору трудные вопросы, чтобы его интерес не угасал! Форум и архивы прошлых обсуждений остаются доступны на сайте издательства до тех пор, пока книга продолжает допечатываться.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Мэттью Фаулер почти 20 лет занимается программной инженерией на различных должностях: от архитектора ПО до директора по инженерии. Он начинал с написания научных программ, затем перешел на сквозную разработку веб-приложений и распределенных систем и в итоге возглавил несколько команд разработчиков и менеджеров, которые занимались созданием интернет-магазина, обслуживающего десятки миллионов пользователей. Проживает в Лексингтоне, штат Массачусетс, с женой Кэти.

1 Первое знакомство с asyncio

Краткое содержание главы

- Что такое библиотека asyncio и какие преимущества она дает.
- Конкурентность, параллелизм, потоки и процессы.
- Глобальная блокировка интерпретатора и создаваемые ей проблемы для конкурентности.
- Как неблокирующие сокеты позволяют добиться конкурентного выполнения в одном потоке.
- Начала конкурентности на основе цикла событий.

Многие программы, особенно в нынешнем мире, где правят бал веб-приложения, сильно зависят от операций ввода-вывода. Это скачивание веб-страницы из интернета, взаимодействие по сети с группой микросервисов или отправка нескольких запросов такой базе данных, как MySQL или Postgres. Выполнение веб-запроса или взаимодействие с микросервисом может занять несколько сотен миллисекунд или даже секунды, если сеть медленная. Запрос к базе данных может занимать много времени, особенно если нагрузка на базу высока или запрос сложный. Веб-серверу иногда приходится обрабатывать одновременно сотни или тысячи запросов.

Попытка выполнить эти запросы ввода-вывода сразу может привести к серьезным проблемам с производительностью. Если отправлять запросы один за другим, как в последовательном приложении, то за-

медление будет нарастать. Например, если мы пишем приложение, которое должно скачать 100 веб-страниц или выполнить 100 запросов к базе данных, и каждое взаимодействие продолжается 1 с, то для завершения приложения потребуется 100 с. Но если воспользоваться конкурентностью и начать все скачивания в одно и то же время, после чего дожидаться результатов, то теоретически все операции можно было бы завершить за 1 с.

Библиотека asyncio впервые появилась в версии Python 3.4 как еще один способ справляться с высокими конкурентными нагрузками, не прибегая к нескольким потокам или процессам. При правильном использовании эта библиотека может значительно повысить производительность и уменьшить потребление ресурсов в приложениях, выполняющих много операций ввода-вывода, поскольку позволяет запускать сразу много таких долго работающих задач.

В этой главе мы познакомимся с основами конкурентности, чтобы лучше понять, как она достигается в Python и библиотеке asyncio. Мы рассмотрим различия между задачами, ограниченными быстродействием процессора и производительностью ввода-вывода, чтобы вы могли понять, какая модель конкурентности лучше отвечает вашим потребностям. Мы также поговорим об основах процессов и потоков и о специфических проблемах, связанных с наличием в Python глобальной блокировки интерпретатора (GIL). Наконец, мы поймем, как использовать *неблокирующий ввод-вывод* совместно с циклом событий и таким образом добиться конкурентности всего в одном процессе и потоке. Это основная модель конкурентности в asyncio.

1.1 Что такое asyncio?

В синхронном приложении код исполняется последовательно. Следующая строка кода выполняется после завершения предыдущей, и в каждый момент времени происходит что-то одно. Эта модель хорошо работает для многих, если не для большинства приложений. Но что, если какая-то одна строка кода занимает слишком много времени? В таком случае весь последующий код должен будет замереть, пока эта строка не соблаговолит завершиться. Многие из нас раньше встречали плохо написанные пользовательские интерфейсы, в которых все поначалу шло хорошо, а потом приложение внезапно зависало, оставляя нас созерцать крутящееся колесико или ждать хоть какого-нибудь ответа. Такие блокировки в приложении оставляют тягостное впечатление у пользователя.

Любая достаточно длительная операция может блокировать приложение, но особенно часто это бывает, когда приложение ждет завершения ввода-вывода. Ввод-вывод выполняют такие устройства, как клавиатура, жесткий диск и, конечно же, сетевая карта. Такие операции ждут ввода от пользователя или получения содержимого

от веб-API. В синхронном приложении мы будем ждать завершения операции и до тех пор ничего не сможем делать. Это ведет к проблемам с производительностью и отзывчивостью, поскольку в каждый момент времени может выполняться только одна длительная операция, а она не дает приложению больше ничего делать.

Один из способов решения этой проблемы – ввести в программу конкурентность. Говоря по-простому, *конкурентность* позволяет одновременно выполнять более одной задачи. Примерами конкурентного ввода-вывода могут служить одновременная отправка нескольких веб-запросов или создание одновременных подключений к веб-серверу.

В Python есть несколько способов организовать такую конкурентность. Одним из последних добавлений в экосистему Python является библиотека асинхронного ввода-вывода *asyncio*. Она позволяет исполнять код в рамках модели асинхронного программирования, т. е. производить сразу несколько операций ввода-вывода, не жертвуя отзывчивостью приложения.

Так что же означают слова «асинхронное программирование»? Что длительную задачу можно выполнять в фоновом режиме отдельно от главного приложения. И система не блокируется в ожидании завершения этой задачи, а может заниматься другими вещами, не зависящими от ее исхода. Затем, по завершении задачи, мы получим уведомление о том, что она все сделала, и сможем обработать результат.

В Python версии 3.4, когда состоялся дебют библиотеки asyncio, она включала декораторы и синтаксис генератора `yield from` для определения сопрограмм. Сопрограмма – это метод, который можно приостановить, если имеется потенциально длительная задача, а затем возобновить, когда она завершится. В Python 3.5 в самом языке была реализована полноценная поддержка сопрограмм и асинхронного программирования, для чего были добавлены ключевые слова `async` и `await`. Этот синтаксис, общий с другими языками программирования, например C и JavaScript, позволяет писать асинхронный код так, что он выглядит как синхронный. Такой асинхронный код проще читать и понимать, поскольку он похож на последовательный код, с которым большинство программистов хорошо знакомо. Библиотека asyncio исполняет сопрограммы асинхронно, пользуясь моделью конкурентности, получившей название *однопоточный цикл событий*.

Название asyncio может навести на мысль, будто библиотека годится только для операций ввода-вывода. Но на самом деле она способна выполнять и операции других типов благодаря взаимодействию с механизмами многопроцессности и многопоточности. Поэтому синтаксис `async` и `await` можно использовать в сочетании с процессами и потоками, что делает соответствующий код понятнее. Следовательно, библиотека пригодна не только для организации конкурентного ввода-вывода, но и для выполнения счетных задач, активно использующих процессор. Чтобы лучше разобраться в том, с какими рабочими нагрузками позволяет справляться asyncio и какая модель

лучше всего подходит для каждого типа конкурентности, рассмотрим различия между операциями, ограниченными производительностью ввода-вывода и быстродействием процессора.

1.2 Что такое ограниченность производительностью ввода-вывода и ограниченность быстродействием процессора

Говоря, что операция ограничена производительностью ввода-вывода или быстродействием процессора, мы имеем в виду фактор, препятствующий более быстрой работе. Это значит, что если увеличить производительность того, что ограничивает операцию, то для ее завершения понадобится меньше времени.

Операция, ограниченная быстродействием процессора (счетная операция), завершилась бы быстрее, будь процессор более мощным, например с частотой 3, а не 2 ГГц. Операция, ограниченная производительностью ввода-вывода, работала бы быстрее, если бы устройство могло обработать больше данных за меньшее время. Для этого можно было бы увеличить пропускную способность сети, заплатив больше денег интернет-провайдеру или поставив более шуструю сетевую карту.

Счетные операции в Python – это обычно вычисления и обработка данных. Примерами могут служить вычисления цифр числа π или применение бизнес-логики к каждому элементу словаря. В случае операции, ограниченной вводом-выводом, мы тратим большую часть времени на ожидание ответа от сети или другого устройства. Примерами могут служить запрос к веб-серверу или чтение файла с жесткого диска.

Листинг 1.1 Операции, ограниченные производительностью ввода-вывода и быстродействием процессора

```
import requests
response = requests.get('https://www.example.com')
items = response.headers.items()
headers = [f'{key}: {header}' for key, header in items]
formatted_headers = '\n'.join(headers)
with open('headers.txt', 'w') as file:
    file.write(formatted_headers)
```

Веб-запрос ограничен производительностью ввода-вывода

Обработка ответа ограничена быстродействием процессора

Конкатенация строк ограничена быстродействием процессора

Запись на диск ограничена производительностью ввода-вывода

Операции, ограниченные производительностью ввода-вывода и быстродействием процессора, обычно сосуществуют бок о бок. Сначала мы выполняем ограниченный производительностью ввода-вывода запрос, чтобы скачать содержимое страницы https://www.example.com. Получив ответ, мы выполняем ограниченный быстродействием процессора цикл форматирования заголовков ответа, в котором они преобразуются в строку и разделяются символами новой строки. Затем мы открываем файл и записываем в него строку – обе операции ограничены производительностью ввода-вывода.

Асинхронный ввод-вывод позволяет приостановить выполнение метода, встретив операцию ввода-вывода; ожидая завершения этой операции, работающей в фоновом режиме, мы можем выполнять какой-нибудь другой код. Это позволяет выполнять одновременно много операций ввода-вывода и тем самым ускорить работу приложения.

1.3 Конкурентность, параллелизм и многозадачность

Чтобы лучше понять, как конкурентность может повысить производительность приложения, важно для начала разобраться в терминологии конкурентного программирования. Мы узнаем, что означает слово «конкурентность» и как в asyncio используется концепция многозадачности для ее достижения. Конкурентность и параллелизм – два понятия, помогающие понять, как осуществляется планирование программы и какие задачи, методы и процедуры приводят весь механизм в действие.

1.3.1 Конкурентность

Говоря, что две задачи выполняются *конкурентно*, мы имеем в виду, что они работают в одно и то же время. Взять, к примеру, пекаря, выпекающего два разных торта. Чтобы их испечь, нужно сначала разогреть духовку. Разогрев может занимать десятки минут – это зависит от духовки и требуемой температуры, но нам необязательно ждать, пока духовка нагреется, поскольку можно в это время заняться другими делами, например смешать муку с сахаром и вбить в тесто яйца. Это можно делать, пока духовка звуковым сигналом не известит нас о том, что нагрелась.

Мы также не хотим связывать себя ограничением – приступать ко второму торту только после готовности первого. Ничто не мешает замесить тесто для одного торта, положить его в миксер и начать готовить вторую порцию, пока первая взбивается. В этой модели мы переключаемся между разными задачами конкурентно. Такое переключение (делать что-то, пока духовка разогревается, переключаться с одного торта на другой) – пример *конкурентного* поведения.

1.3.2 Параллелизм

Хотя конкурентность подразумевает, что несколько задач выполняется одновременно, это еще не значит, что они работают параллельно. Говоря о *параллельной* работе, мы имеем в виду, что две задачи или более не просто чередуются, а выполняются строго в одно и то же время. Возвращаясь к примеру с тортами, представьте, что мы призвали на помощь второго пекаря. В этом случае мы можем трудиться над первым тортом, а помощник будет заниматься вторым. Два человека, готовящих тесто, работают параллельно, потому что одновременно делают два разных дела (рис. 1.1).

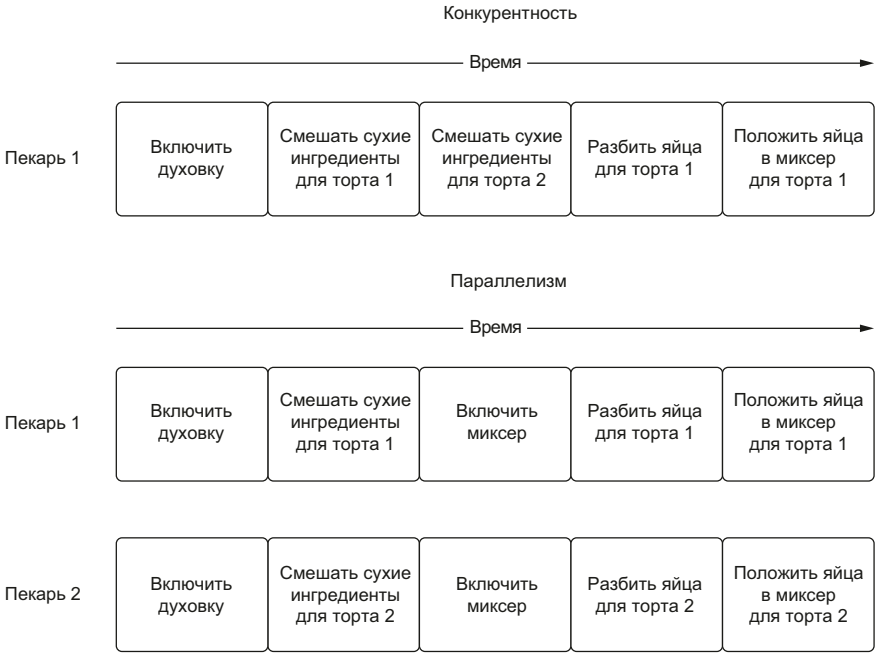

Рис. 1.1 В случае *конкурентности* несколько задач работает в течение одного промежутка времени, но только одна активна в каждый момент. В случае *параллелизма* несколько задач активно одновременно

Теперь переведем это на язык приложений, исполняемых операционной системой. Пусть работают два приложения. В конкурентной системе мы можем переключаться между ними, дав немного поработать сначала одному, а потом другому. Если делать это достаточно быстро, создается впечатление, что два дела делаются одновременно. В параллельной системе два приложения работают действительно одновременно, т. е. оба активны в одно и то же время.

Понятия конкурентности и параллелизма похожи (рис. 1.2), и различить их бывает затруднительно, но важно хорошо понимать, чем они отличаются.

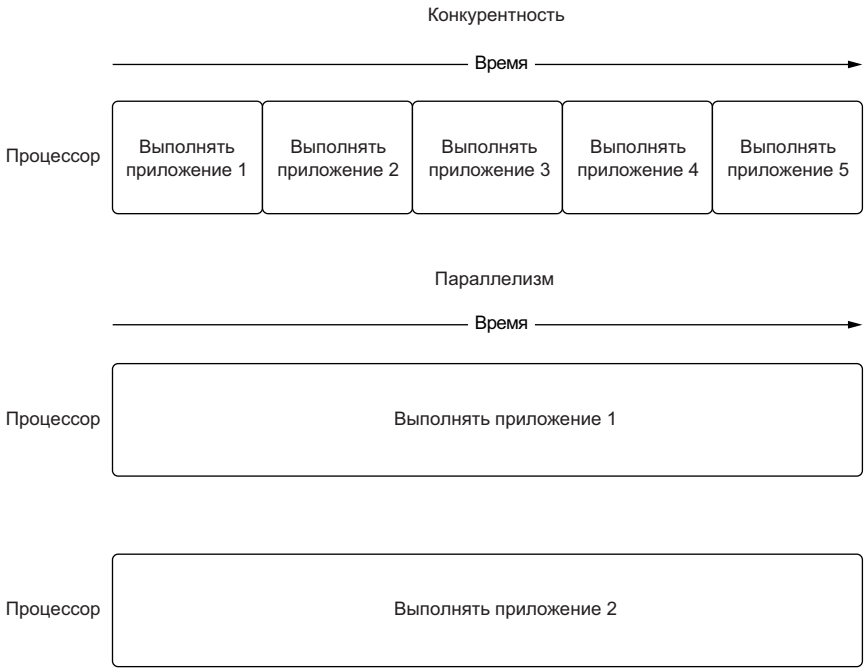

Рис. 1.2 В случае *конкурентности* мы переключаемся между двумя приложениями. В случае *параллелизма* мы активно выполняем два приложения одновременно

1.3.3 Различие между конкурентностью и параллелизмом

Конкурентность возможна, когда несколько задач может работать независимо друг от друга. Конкурентность можно организовать, имея процессор всего с одним ядром, применив *вытесняющую многозадачность* (определяется в следующем разделе) для переключения между задачами. С другой стороны, параллелизм означает, что мы должны выполнять две задачи или более строго одновременно. На машине с одним ядром это невозможно, необходимо иметь процессор с несколькими ядрами.

Параллелизм подразумевает конкурентность, но обратное верно не всегда. Многопоточное приложение, работающее на многоядерной машине, является и конкурентным, и параллельным. В этом случае имеется несколько задач, работающих одновременно, и два ядра, независимо исполняющих код этих задач. Но многозадачность допускает также наличие нескольких задач, работающих конкурентно, но так, что в каждый момент времени выполняется только одна.

1.3.4 Что такое многозадачность

В современном мире многозадачность встречается повсеместно. Мы готовим завтрак, а пока в чайнике закипает вода, кому-то звоним или отвечаем на текстовое сообщение. А пока электричка везет нас на работу, читаем книгу. В этом разделе мы обсудим два основных вида многозадачности: *вытесняющую* и *невытесняющую*, или *кооперативную*.

Вытесняющая многозадачность

В этой модели мы позволяем операционной системе решить, как переключаться между выполняемыми задачами с помощью процедуры *квантования времени*. Когда операционная система переключает задачи, мы говорим, что имеет место *вытеснение*.

Как устроен этот механизм, зависит от операционной системы. Обычно для этого используется либо несколько потоков, либо несколько процессов.

Кооперативная многозадачность

В этой модели мы не полагаемся для переключения между задачами на операционную систему, а явно определяем в коде приложения точки, где можно уступить управление другой задаче. Исполняемые задачи *кооперируются*, т. е. говорят приложению: «Я сейчас на время приостановлюсь, а ты можешь пока выполнять другие задачи».

1.3.5 Преимущества кооперативной многозадачности

В asyncio для организации конкурентности используется кооперативная многозадачность. Когда приложение доходит до точки, в которой может подождать результата, мы явно помечаем это в коде. Поэтому другой код может работать, пока мы ждем получения результата, вычисляемого в фоновом режиме. Как только вычисление результата завершится, мы «просыпаемся» и возобновляем задачу. Это является формой конкурентности, потому что несколько задач может работать одновременно, но – и это очень важно – не параллельно, так как их выполнение чередуется.

У кооперативной многозадачности есть ряд преимуществ перед вытесняющей. Во-первых, кооперативная многозадачность потребляет меньше ресурсов. Когда операционной системе нужно переключиться между потоками или процессами, мы говорим, что имеет место *контекстное переключение*. Это трудоемкая операция, потому что операционная система должна сохранить всю информацию о работающем процессе или потоке, чтобы потом его можно было возобновить.

Второе преимущество – *гранулярность*. Операционная система приостанавливает поток или процесс в соответствии со своим алго-

ритмом планирования, но выбранный для этого момент не всегда оптимален. В случае кооперативной многозадачности мы явно помечаем точки, в которых приостановить задачу наиболее выгодно. Это дает выигрыш в эффективности, потому что мы переключаем задачи именно в тот момент, когда это нужно. А теперь, когда мы разобрались в понятиях конкурентности, параллелизма и многозадачности, посмотрим, как они реализуются в Python с помощью потоков и процессов.

1.4 Процессы, потоки, многопоточность и многопроцессность

Чтобы лучше понять, как работает конкурентность в Python, нужно сначала разобраться с потоками и процессами. Затем мы поговорим о том, как использовать их для организации конкурентной работы с помощью многопоточности и многозадачности. Начнем с определения процесса и потока.

1.4.1 Процесс

Процессом называется работающее приложение, которому выделена область памяти, недоступная другим приложениям. Пример создания Python-процесса – запуск простого приложения «hello world» или ввод слова `python` в командной строке для запуска цикла REPL (цикл чтения–вычисления–печати).

На одной машине может работать несколько процессов. Если машина оснащена процессором с несколькими ядрами, то несколько процессов могут работать одновременно. Если процессор имеет только одно ядро, то все равно можно выполнять несколько приложений конкурентно, но уже с применением квантования времени. При использовании квантования операционная система будет автоматически вытеснять работающий процесс по истечении некоторого промежутка времени и передавать процессор другому процессу. Алгоритмы, определяющие, в какой момент переключать процессы, зависят от операционной системы.

1.4.2 Поток

Потоки можно представлять себе как облегченные процессы. Кроме того, это наименьшая единица выполнения, которая может управляться операционной системой. У потоков нет своей памяти, они пользуются памятью создавшего их процесса. Потоки ассоциированы с процессом, создавшим их. С каждым процессом всегда ассоциирован по меньшей мере один поток, обычно называемый *главным*. Процесс может создавать дополнительные потоки, которые обычно

называются *рабочими* или *фоновыми*. Эти потоки могут конкурентно выполнять другую работу наряду с главным потоком. Потоки, как и процессы, могут работать параллельно на многоядерном процессе, и операционная система может переключаться между ними с помощью квантования времени. Обычное Python-приложение создает процесс и главный поток, который отвечает за его выполнение.

Листинг 1.2 Процессы и потоки в простом Python-приложении

```
import os
import threading

print(f'Исполняется Python-процесс с идентификатором: {os.getpid()}')

total_threads = threading.active_count()
thread_name = threading.current_thread().name

print(f'В данный момент Python исполняет {total_threads} поток(ов)')
print(f'Имя текущего потока {thread_name}')
```

На рис. 1.3 схематически изображен процесс, соответствующий листингу 1.2. Мы написали простое приложение, демонстрирующее работу главного потока. Сначала мы получаем уникальный идентификатор процесса и печатаем его, чтобы доказать, что действительно имеется работающий процесс. Затем получаем счетчик активных потоков и имя текущего потока, чтобы доказать, что действительно работает один поток – главный. При каждом выполнении этой программы идентификатор процесса будет разным, но всегда выводятся строки вида:

```
Исполняется Python-процесс с идентификатором: 98230
В данный момент Python исполняет  1 поток(ов)
Имя текущего потока MainThread
```

Рис. 1.3 Процесс с одним главным потоком, читающим данные из памяти

Процессы могут порождать дополнительные потоки, разделяющие память со своим процессом-родителем. Они могут конкурентно выполнять другую работу, это называется *многопоточностью*.

Листинг 1.3 Создание многопоточного Python-приложения

```
import threading

def hello_from_thread():
    print(f'Привет от потока {threading.current_thread()}!')

hello_thread = threading.Thread(target=hello_from_thread)
hello_thread.start()

total_threads = threading.active_count()
thread_name = threading.current_thread().name

print(f'В данный момент Python выполняет {total_threads} поток(ов)')
print(f'Имя текущего потока {thread_name}')
hello_thread.join()
```

На рис. 1.4 схематически изображен процесс и потоки, соответствующие листингу 1.3. Мы написали метод, печатающий имя текущего потока, а затем создали поток, исполняющий этот метод. После этого вызывается метод потока `start`, запускающий поток. А в конце вызывается метод `join`, который приостанавливает программу, до тех пор пока указанный поток не завершится. Этот код печатает такие сообщения:

```
Привет от потока <Thread(Thread-1, started 123145541312512)>!
В данный момент Python выполняет 2 поток(ов)
Имя текущего потока MainThread
```

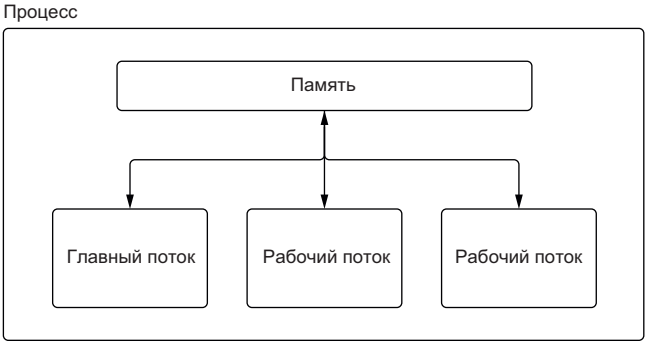

Рис. 1.4 Многопоточная программа с двумя рабочими и одним главным потоком. Все они разделяют общую память с процессом

Отметим, что при выполнении этой программы сообщения «Привет от потока» и «В данный момент Python выполняет 2 поток(ов)» могут быть напечатаны на одной строке. Это *состояние гонки*, мы будем говорить о нем в следующем разделе и подробнее в главах 6 и 7.

Многопоточные приложения – обычный способ реализации конкурентности во многих языках программирования. Но в Python есть несколько препятствий на пути организации конкурентности с по-

мощью потоков. Многопоточность полезна только для задач, ограниченных производительностью ввода-вывода, потому что нам мешает глобальная блокировка интерпретатора, о которой пойдет речь в главе 1.5.

Многопоточность не единственный способ добиться конкурентности. Можно вместо потоков создать несколько конкурентных процессов, это называется *многопроцессностью*. В таком случае родительский процесс создает один или более дочерних процессов, которыми управляет, а затем распределяет между ними работу.

В Python для этой цели имеется модуль `multiprocessing`. Его API похож на API модуля `threading`. Сначала создается процесс, при этом передается функция `target`. Затем вызывается метод `start`, чтобы начать выполнение процесса, и в конце – метод `join`, чтобы дождаться его завершения.

Листинг 1.4 Создание нескольких процессов

```
import multiprocessing
import os

def hello_from_process():
    print(f'Привет от дочернего процесса {os.getpid()}!')

if __name__ == '__main__':
    hello_process = multiprocessing.Process(target=hello_from_process)
    hello_process.start()

    print(f'Привет от родительского процесса {os.getpid()}')

    hello_process.join()
```

На рис. 1.5 схематически показан процесс и потоки для листинга 1.4. Мы создаем один дочерний процесс, который печатает свой идентификатор, а также печатаем идентификатор родительского процесса, чтобы доказать, что работают два разных процесса. Многопроцессность обычно предпочтительна, когда имеется счетная задача.

Многопоточность и многопроцессность могут показаться волшебной палочкой, обеспечивающей конкурентность в Python. Однако развернуться во всю мощь этим моделям мешает одна деталь реализации Python – глобальная блокировка интерпретатора.

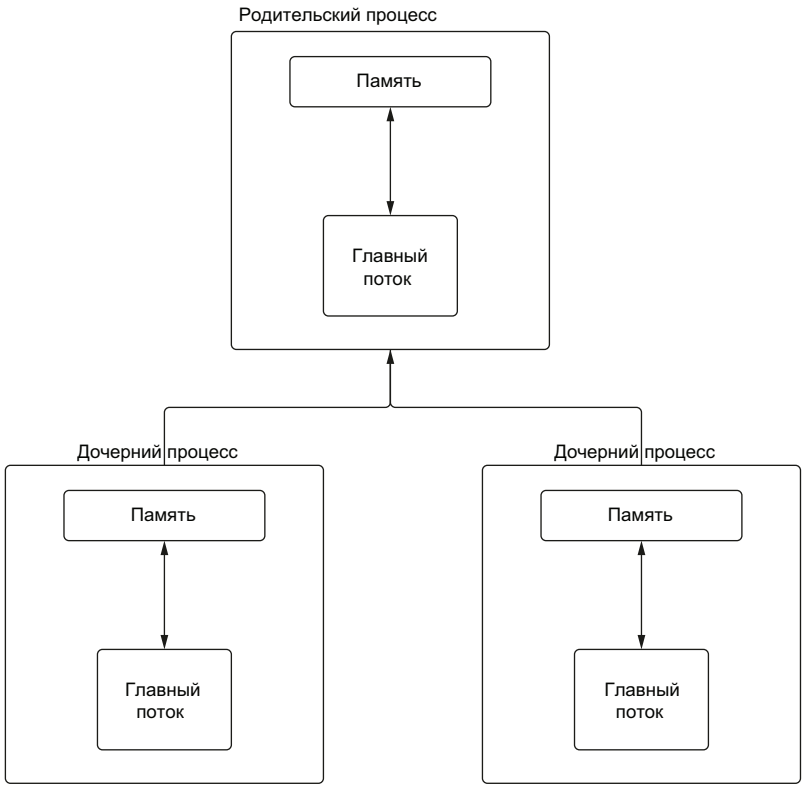

Рис. 1.5 Приложение, в котором имеется один родительский процесс и два дочерних

1.5 Глобальная блокировка интерпретатора

Глобальная блокировка интерпретатора (*global interpreter lock* – GIL) – тема, вызывающая споры в сообществе Python. Говоря кратко, GIL не дает Python-процессу исполнять более одной команды байт-кода в каждый момент времени. Это означает, что, даже если имеется несколько потоков на многоядерной машине, интерпретатор сможет в каждый момент исполнять только один поток, содержащий написанный на Python код. В мире, где процессоры имеют несколько ядер, это создает серьезную проблему для разработчиков на Python, желающих воспользоваться многопоточностью для повышения производительности приложений.

ПРИМЕЧАНИЕ Многопроцессные приложения могут конкурентно выполнять несколько команд байт-кода, потому что у каждого Python-процесса своя собственная GIL.

Так для чего же нужна GIL? Ответ кроется в том, как CPython управляет памятью. В CPython память управляется в основном с помощью *подсчета ссылок*. То есть для каждого объекта Python, например целого числа, словаря или списка, подсчитывается, сколько объектов в данный момент используют его. Когда объект перестает быть нужным кому-то, счетчик ссылок на него уменьшается, а когда кто-то новый обращается к нему, счетчик ссылок увеличивается. Если счетчик ссылок обратился в нуль, значит, на объект никто не ссылается, поэтому его можно удалить из памяти.

Что такое CPython?

CPython – это *эталонная реализация* Python, т. е. стандартная реализация языка, которая используется как *эталон* правильного поведения. Существуют и другие реализации, например Jython, работающая под управлением виртуальной машины Java, или IronPython, предназначенная для .NET Framework.

Конфликт потоков возникает из-за того, что интерпретатор CPython не является *потокобезопасным*. Это означает, что если два или более потоков модифицируют разделяемую переменную, то ее конечное состояние может оказаться неожиданным, поскольку зависит от порядка доступа к переменной со стороны потоков. Эта ситуация называется *состоянием гонки*. Состояния гонки могут возникать, когда два потока одновременно обращаются к одному объекту Python.

Как показано на рис. 1.6, если два потока одновременно увеличивают счетчик ссылок, то может случиться, что счетчик обнулится, хотя объект еще используется. Результатом станет крах приложения при попытке прочитать данные из освобожденной памяти.

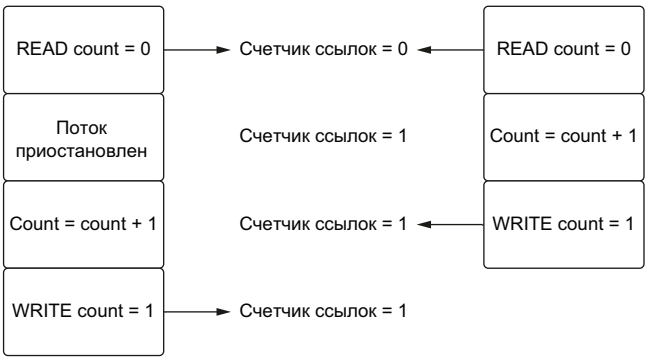

Рис. 1.6 Состояние гонки, возникающее, когда два потока одновременно пытаются увеличить счетчик ссылок. Вместо ожидаемого значения счетчика 2 мы получили значение 1

Для демонстрации влияния GIL на многопоточное программирование рассмотрим счетную задачу вычисления *n*-го числа Фибоначчи. Мы будем использовать довольно медленный алгоритм, чтобы продемонстрировать операции, занимающие много времени. В правильном решении для повышения производительности следовало бы использовать технику запоминания или другой математический метод.

Листинг 1.5 Генерирование последовательности Фибоначчи и его хронометраж

```
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

    print(f'fib({number}) равно {fib(number)}')

def fibs_no_threading():
    print_fib(40)
    print_fib(41)

start = time.time()

fibs_no_threading()

end = time.time()

print(f'Время работы {end - start:.4f} с.')
```

В этой реализации используется рекурсия, поэтому алгоритм получился медленным, он занимает время *O*(*2^N*). Если нужно напечатать два числа Фибоначчи, то можно просто вычислить их синхронно и замерить время вычисления, как показано в листинге выше.

В зависимости от быстродействия процессора результаты хронометража могут быть различны, но результат будет примерно таким, как показано ниже.

```
fib(40) равно 63245986
fib(41) равно 102334155
Время работы 65.1516 с.
```

Вычисление долгое, но обращения к функции `print_fib` не зависят друг от друга. Это значит, что их можно поместить в разные потоки, которые теоретически могли бы исполняться разными процессорными ядрами, что ускорило бы работу приложения.

Листинг 1.6 Многопоточное вычисление последовательности чисел Фибоначчи

```
import threading
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

def fibs_with_threads():
    fortieth_thread = threading.Thread(target=print_fib, args=(40,))
    forty_first_thread = threading.Thread(target=print_fib, args=(41,))

    fortieth_thread.start()
    forty_first_thread.start()

    fortieth_thread.join()
    forty_first_thread.join()

start_threads = time.time()

fibs_with_threads()

end_threads = time.time()

print(f'Многопоточное вычисление заняло {end_threads - start_threads:.4f} с.')
```

Здесь мы создали два потока, один из которых вычисляет `fib(40)`, а другой `fib(41)`, и запустили их конкурентно, вызвав метод `start()` для каждого. Затем мы дважды вызвали метод `join()`, благодаря чему главная программа будет ждать завершения потоков. Учитывая, что вычисления `fib(40)` и `fib(41)` начались одновременно и выполняются конкурентно, можно было бы ожидать разумного ускорения работы, однако даже на многоядерной машине мы видим такой результат:

```
fib(40) равно 63245986
fib(41) равно 102334155
Многопоточное вычисление заняло 66.1059 с.
```

Многопоточная версия работала почти столько же времени. На самом деле даже чуть дольше! Это все из-за GIL и накладных расходов на создание и управление потоками. Да, потоки выполняются конкурентно, но в каждый момент времени только одному из них разрешено выполнять Python-код. А второй поток вынужден ждать завершения первого, что сводит на нет весь смысл нескольких потоков.

1.5.1 Освобождается ли когда-нибудь GIL?

Предыдущий пример поднимает вопрос, возможна ли вообще многопоточная конкурентность в Python, коль скоро GIL запрещает одновременное выполнение двух строк Python-кода? Но на наше счастье GIL не удерживается постоянно, что открывает возможность использования нескольких потоков.

Глобальная блокировка интерпретатора освобождается на время выполнения операций ввода-вывода. Это позволяет использовать потоки для конкурентного выполнения ввода-вывода, но не для выполнения счетного кода, написанного на Python (есть исключения, когда GIL все же освобождается на время выполнения счетных задач, и мы обсудим их в следующей главе). Для иллюстрации рассмотрим чтение кода состояния веб-страницы.

Листинг 1.7 Синхронное чтение кода состояния

```
import time
import requests


def read_example() -> None:
    response = requests.get('https://www.example.com')
    print(response.status_code)


sync_start = time.time()

read_example()
read_example()

sync_end = time.time()

print(f'Синхронное выполнение заняло{sync_end - sync_start:.4f} с.')
```

Здесь мы дважды скачиваем содержимое страницы *example.com* и печатаем код состояния. Результат зависит от скорости сетевого подключения и нашего местоположения, но в целом будет примерно таким:

```
200
200
Синхронное выполнение заняло 0.2306 с.
```

Теперь у нас есть эталон для сравнения – время работы синхронной версии, – и можно написать многопоточную версию. В ней мы создадим по одному потоку для каждого запроса к *example.com* и запустим их конкурентно.

```
import time
import threading
import requests
```

```
def read_example() -> None:
    response = requests.get('https://www.example.com')
    print(response.status_code)


thread_1 = threading.Thread(target=read_example)
thread_2 = threading.Thread(target=read_example)

thread_start = time.time()

thread_1.start()
thread_2.start()

print('Все потоки работают!')

thread_1.join()
thread_2.join()

thread_end = time.time()

print(f'Многопоточное выполнение заняло {thread_end - thread_start:.4f} с.')
```

В результате выполнения этой программы мы увидим следующие строки, где время, как и раньше, зависит от сетевого подключения и местоположения:

```
Все потоки работают!
200
200
Многопоточное выполнение заняло 0.0977 с.
```

Это примерно в два раза быстрее первоначальной версии, в которой потоки не использовались, поскольку теперь два запроса выполняются приблизительно в одно и то же время! Разумеется, результат зависит от скорости подключения к интернету и от характеристик машины, но порядок цифр именно такой.

Так почему же GIL освобождается при вводе-выводе, но не освобождается для счетных задач? Все дело в системных вызовах, которые выполняются за кулисами. В случае ввода-вывода низкоуровневые системные вызовы работают за пределами среды выполнения Python. Это позволяет освободить GIL, потому что код операционной системы не взаимодействует напрямую с объектами Python. GIL захватывается снова, только когда полученные данные переносятся в объект Python. Стало быть, на уровне ОС операции ввода-вывода выполняются конкурентно. Эта модель обеспечивает конкурентность, но не параллелизм. В других языках, например Java или C++, на многоядерных машинах можно организовать истинный параллелизм, потому что никакой GIL нет и код может выполняться строго одновременно. Но в Python лучшее, на что можно рассчитывать, – конкурентность операций ввода-вывода, поскольку в любой момент может выполняться только один кусок написанного на Python кода.

1.5.2 Asyncio и GIL

В asyncio используется тот факт, что операции ввода-вывода освобождают GIL, что позволяет реализовать конкурентность даже в одном потоке. При работе с asyncio мы создаем объекты *сопрограмм*. Сопрограмму можно представлять себе как облегченный поток. Точно так же, как может быть несколько потоков, работающих одновременно и исполняющих разные операции ввода-вывода, так может существовать много сопрограмм, работающих бок о бок. Ожидая завершения сопрограмм, занимающихся вводом-выводом, мы можем выполнять другой Python-код, получая таким образом конкурентность. Важно отметить, что asyncio не обходит GIL, мы по-прежнему ограничены ей. Если имеется счетная задача, то для ее конкурентного выполнения все равно нужно заводить отдельный процесс (и в asyncio есть для этого средства), иначе производительность снизится. Теперь, когда мы знаем, как организовать конкурентный ввод-вывод в одном потоке, посмотрим, как это работает на примере неблокирующих сокетов.

1.6 Как работает однопоточная конкурентность

В предыдущем разделе мы рассмотрели многопоточность как механизм конкурентного выполнения операций ввода-вывода. Однако для достижения такого вида конкурентности заводить несколько потоков ни к чему. Все это можно сделать в рамках одного процесса и одного потока. При этом мы воспользуемся тем фактом, что на уровне ОС операции ввода-вывода могут завершаться конкурентно. Чтобы лучше понять, как это возможно, рассмотрим подробнее, как работают сокеты и конкретно неблокирующие сокеты.

1.6.1 Что такое сокет?

Сокет – это низкоуровневая абстракция отправки и получения данных по сети. Именно с ее помощью производится обмен данными между клиентами и серверами. Сокеты поддерживают две основные операции: отправку и получение байтов. Мы записываем байты в сокет, затем они передаются по адресу назначения, чаще всего на какой-то сервер. Отправив байты, мы ждем, пока сервер пришлет ответ в наш сокет. Когда байты окажутся в сокете, мы сможем прочитать результат.

Низкоуровневую концепцию сокетов проще понять, если представлять их как почтовые ящики. Вы можете положить письмо в почтовый ящик, почтальон заберет его и доставит в почтовый ящик получателя. Получатель откроет его и достанет ваше письмо. В зависимости от содержания письма получатель может отправить письмо

в ответ. Здесь письмо является аналогом данных или байтов, которые мы хотим отправить. Можно рассматривать помещение письма в почтовый ящик как запись байтов в сокет, а извлечение его из ящика – как чтение байтов из сокета. Почтальон тогда является аналогом механизма передачи через интернет, который маршрутизирует данные до адреса назначения.

Если нужно получить содержимое страницы *example.com*, то мы открываем сокет, подключенный к серверу *example.com*. Затем записываем в сокет запрос и ждем ответа от сервера, в данном случае HTML-кода веб-страницы. На рис. 1.7 показан поток байтов между клиентом и сервером.

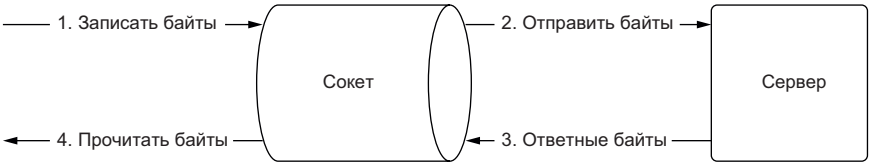

Рис. 1.7 Запись байтов в сокет и чтение байтов из сокета

По умолчанию сокеты *блокирующие*. Это значит, что на все время ожидания ответа от сервера приложение приостанавливается или *блокируется*. Следовательно, оно не может ничего делать, пока не придут данные от сервера или произойдет ошибка или случится тайм-аут.

На уровне операционной системы эта блокировка ни к чему. Сокеты могут работать в *неблокирующем режиме*, когда мы просто начинаем операцию чтения или записи и забываем о ней, а сами занимаемся другими делами. Но позже операционная система уведомляет нас о том, что байты получены, и в этот момент мы можем уделить им внимание. Это позволяет приложению не ждать прихода байтов, а делать что-то полезное. Для реализации такой схемы применяются различные системы уведомления с помощью событий, разные в разных ОС. Библиотека asyncio абстрагирует различия между системами уведомления, а именно:

- *kqueue* – FreeBSD и MacOS;
- *epoll* – Linux;
- *IOCP* (порт завершения ввода-вывода) – Windows.

Эти системы наблюдают за неблокирующими сокетами и уведомляют нас, когда с сокетом можно начинать работу. Именно они лежат в основе модели конкурентности в asyncio. В этой модели имеется только один поток, исполняющий Python-код. Встретив операцию ввода-вывода, интерпретатор передает ее на попечение системы уведомления, входящей в состав ОС. Совершив этот акт, поток Python волен исполнять другой код или открыть другие неблокирующие сокеты, о которых позаботится ОС. По завершении операции система «пробуждает» задачу, ожидающую результата, после чего выполня-

ется код, следующий за этой операцией. Эта схема изображена на рис. 1.8, где имеется несколько операций с разными сокетами.

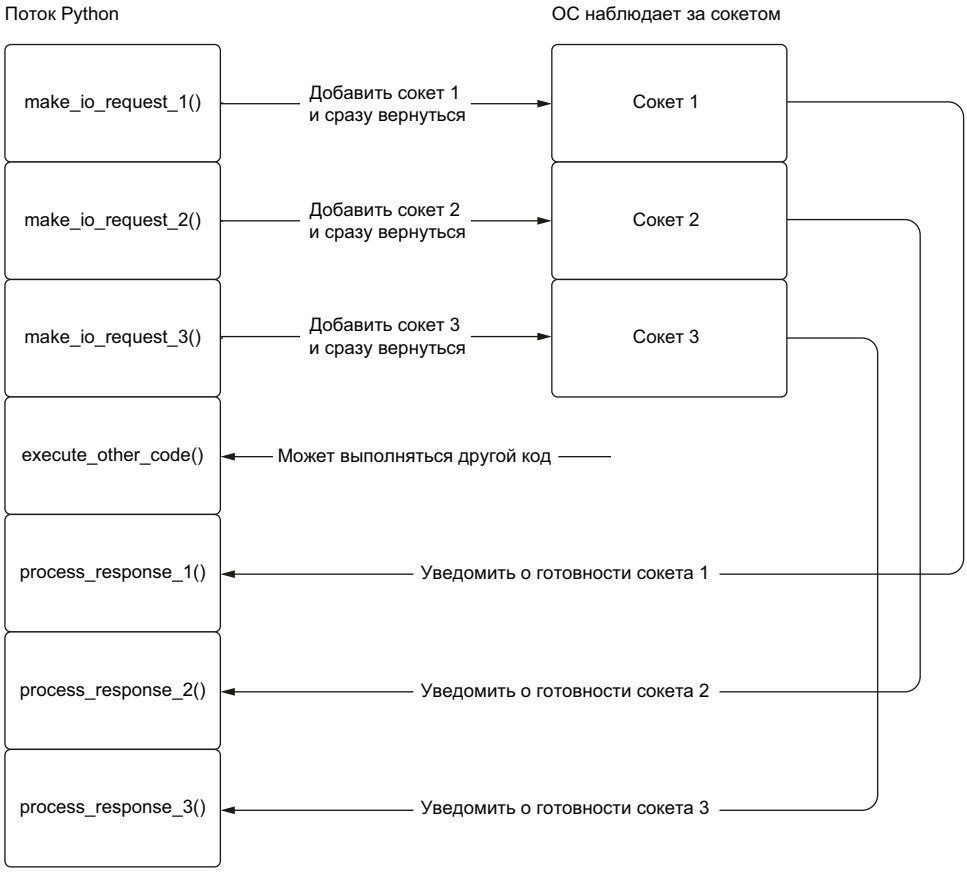

Рис. 1.8 После записи запроса в неблокирующий сокет управление возвращается немедленно, а ОС начинает наблюдать за данными в сокете. Поэтому функция `execute_other_code()` начинает выполняться сразу, не дожидаясь завершения операции ввода-вывода. Когда операция завершится, мы получим уведомление и сможем обработать ответ

Но как теперь отличить задачи, ожидающие завершения ввода-вывода, от тех, которые просто выполняют Python-код и ничего не ждут? Ответ дает конструкция под названием «цикл событий».

1.7 Как работает цикл событий

Цикл событий – сердце любого приложения asyncio. Этот паттерн проектирования встречается во многих системах и был придуман уже довольно давно. Используя в браузере JavaScript для отправки асин-

хронного запроса, вы создаете задачу, управляемую циклом событий. В GUI-приложениях Windows за кулисами используются так называемые циклы обработки сообщений; это основной механизм обработки таких событий, как нажатие клавиш, он позволяет одновременно отрисовывать интерфейс и реагировать на действия пользователя.

По сути своей цикл событий очень прост. Мы создаем очередь, в которой хранится список событий или сообщений, а затем входим в бесконечный цикл, где обрабатываем сообщения по мере их поступления. В Python базовый цикл событий мог бы выглядеть следующим образом:

```
from collections import deque

messages = deque()

while True:
    if messages:
        message = messages.pop()
        process_message(message)
```

В asyncio цикл событий управляет очередью задач, а не сообщений. Задача – это обертка вокруг сопрограммы. Сопрограмма может приостановить выполнение, встретив операцию ввода-вывода, и дать циклу событий возможность выполнить другие задачи, которые не ждут завершения ввода-вывода.

Создавая цикл событий, мы создаем пустую очередь задач. Затем добавляем в эту очередь задачи для выполнения. На каждой итерации цикла проверяется, есть ли в очереди готовая задача, и если да, то она выполняется, пока не встретит операцию ввода-вывода. В этот момент задача приостанавливается, и мы просим операционную систему наблюдать за ее сокетами. А сами тем временем переходим к следующей готовой задаче. На каждой итерации проверяется, завершилась ли какая-нибудь операция ввода-вывода; если да, то ожидавшие ее завершения задачи пробуждаются и им предоставляется возможность продолжить работу. Эта идея иллюстрируется на рис. 1.9: главный поток поставляет задачи циклу событий, а тот их выполняет.

Для конкретики представим, что имеется три задачи, каждая из которых отправляет асинхронный веб-запрос. Допустим, что на этапе инициализации они выполняют некоторый счетный код, затем посылают веб-запрос, а по его завершении обрабатывают результат, что снова требуется счет. На псевдокоде это выглядит так:

```
def make_request():
    cpu_bound_setup()
    io_bound_web_request()
```

```
    cpu_bound_postprocess()

task_one = make_request()
task_two = make_request()
task_three = make_request()
```

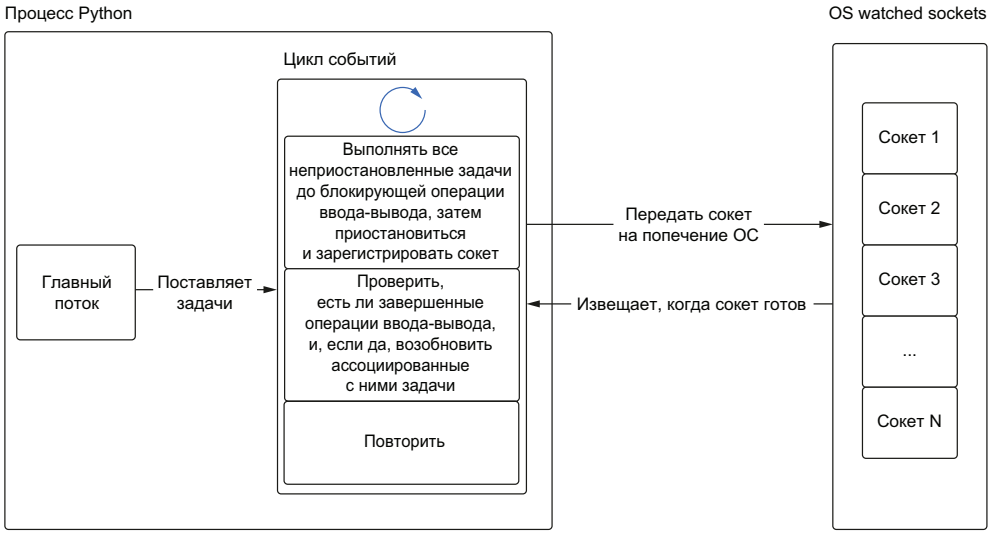

Рис. 1.9 Пример потока, поставляющего задачи циклу событий

Все три задачи вначале выполняют счетные операции, а поскольку поток всего один, то первая задача начинает работать, а остальные две ждут. Закончив счетную часть, задача 1 встречает операцию ввода-вывода и приостанавливается со словами: «Я жду завершения ввода-вывода, пусть поработают другие». После этого начинает работать задача 2 и, встретив операцию ввода-вывода, тоже приостанавливается. В этот момент обе задачи, 1 и 2, ждут завершения ввода-вывода, так что может приступить к работе задача 3.

Теперь допустим, что пока задача 3 ждет завершения своей операции ввода-вывода, пришел ответ на веб-запрос от задачи 1. Операционная система уведомляет нас о том, что ввод-вывод завершен. Мы можем возобновить задачу 1, пока задачи 2 и 3 ждут.

На рис. 1.10 показан поток выполнения, соответствующий только что описанному псевдокоду. Глядя на диаграмму слева направо, мы видим, что в каждый момент времени работает только один счетный кусок кода, но при этом конкурентно выполняются одна или две операции ввода-вывода. Именно из-за такого перекрытия ожидания ввода-вывода asyncio и достигает экономии во времени.

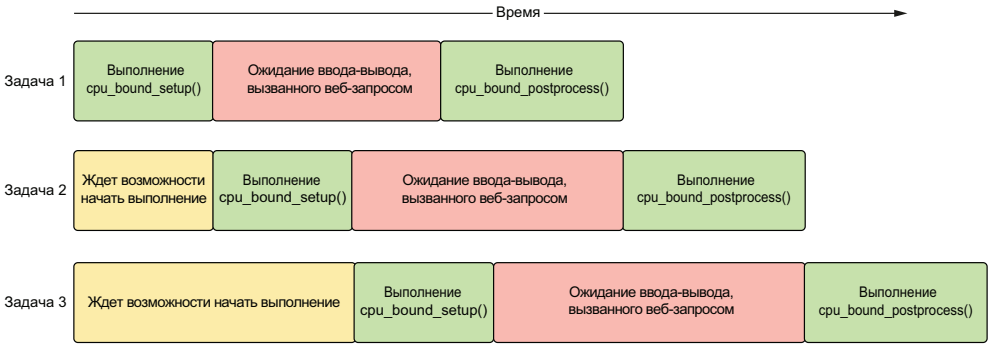

Рис. 1.10 Конкурентное выполнение нескольких задач с операциями ввода-вывода

Резюме

- Ограниченной быстродействием процессора (счетной) называется работа, потребляющая в основном ресурсы процессора, а ограниченной производительностью ввода-вывода – работа, потребляющая в основном ресурсы сети или других устройств ввода-вывода. Главная задача библиотеки asyncio – обеспечить конкурентность задач, ограниченных производительностью ввода-вывода, однако она также предлагает API для организации конкурентности счетных задач.
- Процессы и потоки – основные единицы конкурентности на уровне операционной системы. Процессы можно использовать для рабочих нагрузок, ограниченных как производительностью ввода-вывода, так и быстродействием процессора, а потоки в Python (обычно) – только для эффективного управления задачами, ограниченными производительностью ввода-вывода, потому что GIL не дает выполнять код параллельно.
- Мы видели, что в случае неблокирующих сокетов можно не приостанавливать приложение на время ожидания данных, а попросить операционную систему уведомить нас, когда данные поступят. Именно это позволяет asyncio организовать конкурентность в одном потоке.
- Мы познакомились с циклом событий, лежащим в основе приложений asyncio. В бесконечном цикле событий исполняются счетные задачи, а задачи, ожидающие ввода-вывода, приостанавливаются.