ChatGPT

Многопоточность в Python: продвинутый разбор и внутреннее устройство

1. Потоки в Python: использование `threading.Thread`

В Python модуль `threading` предоставляет класс `Thread` для создания и управления потоками (threads). **Поток** – это отдельная нить исполнения внутри процесса. Каждый поток запускается вызовом метода `start()` и выполняет целевую функцию, указанную при создании. Когда поток завершается, основной поток программы может дождаться его окончания с помощью метода `join()`.

Принцип работы: Под капотом `threading.Thread` в CPython использует *реальные* потоки ОС (например, POSIX threads на Linux). Это означает, что каждый Python-поток соответствует **планируемому ядром потоку**. Однако, из-за **GIL** (Global Interpreter Lock, обсудим подробнее далее) одновременно исполнять байт-код Python может только один поток, поэтому прироста в использовании CPU-ядер для чисто вычислительных задач потоки не дают. Тем не менее, для операций ввода-вывода и ожидания (I/O) потоки полезны – один поток может ожидать операции, освобождая возможность другим выполняться.

Методы потока: - `start()` – запускает поток, вызывая указанную целевую функцию в новом потоке. После `start()` поток переходит в состояние выполнения (в параллель с основным потоком). - `join(timeout=None)` – приостановит выполнение текущего (основного) потока до завершения потока, у которого вызван join. Основной поток «ждёт» поток, обычно для того чтобы убедиться, что все фоновые задачи завершены перед выходом программы. Если указать `timeout`, основной поток будет ждать не дольше указанного времени.

Ниже приведён пример, демонстрирующий создание и запуск нескольких потоков, а также использование `join()` для ожидания их завершения:

```
import threading
import time

def worker(num):
    """Функция, которую будет выполнять поток."""
    print(f"Поток {num} начал работу")
    time.sleep(1)  # имитируем какую-то работу, например, задержку
    print(f"Поток {num} завершил работу")

# Создание и запуск 5 потоков
threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i,))
    t.start()               # запускаем поток (выполнится worker(i))
    threads.append(t)
```

```
# Ожидание завершения всех потоков
for t in threads:
    t.join()
print("Все потоки завершены")
```

Вывод (порядок может различаться):

```
Поток 0 начал работу
Поток 1 начал работу
Поток 2 начал работу
Поток 3 начал работу
Поток 4 начал работу
Поток 0 завершил работу
Поток 1 завершил работу
Поток 2 завершил работу
Поток 3 завершил работу
Поток 4 завершил работу
Все потоки завершены
```

В этом примере потоки запускаются почти одновременно. Благодаря вызовам `time.sleep(1)` все потоки приостанавливаются на секунду (при этом GIL освобождается, позволяя другим потокам выполняться, как обсудим позже). Метод `join()` в конце гарантирует, что основной поток дождётся окончания всех запущенных потоков перед тем, как выводить сообщение о завершении работы.

Обратите внимание: если не вызывать `join()`, основной поток может завершить программу раньше, чем завершатся фоновые потоки. **Важно**: при нормальных (не daemon) потоках Python дождётся их завершения при выходе, но логика программы может нарушиться без явного упорядочивания через `join()`.

2. Конкурентное выполнение с `concurrent.futures`: `as_completed()` vs `map()`

Для управления группой потоков Python предоставляет высокоуровневый интерфейс **пулов потоков** через модуль `concurrent.futures`. Класс `ThreadPoolExecutor` позволяет запускать функции в пуле потоков, абстрагируя создание и уничтожение объектов `Thread`.

Основные способы использования пула: - `Executor.map(func, *iterables)` – применяет функцию `func` к каждому элементу из последовательности (или нескольких последовательностей, подобно встроенной `map`). Возвращает итератор результатов **в том же порядке, что и входные данные**. - `concurrent.futures.as_completed(futures)` – принимает список объектов `Future` (результаты, ассоциированные с асинхронно выполняющимися задачами) и позволяет итерироваться по ним **в порядке завершения задач** (кто завершился раньше, тот возвращается раньше).

Отличие `map()` **от** `as_completed()`:
`Executor.map` упрощает получение результатов параллельных задач, но она будет ждать самый медленный поток чтобы вернуть результаты по порядку. В то же время `as_completed` позволяет обрабатывать результаты сразу по готовности каждой задачи, не дожидаясь остальных.

Например, запустим несколько задач, имитирующих разное время работы:

```
import concurrent.futures, time

def sleep_and_return(name, sec):
    time.sleep(sec)
    return name

tasks = [("Task1", 3), ("Task2", 1), ("Task3", 2)]

# Используем as_completed:
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    futures = [executor.submit(sleep_and_return, name, t) for name, t in
tasks]
    for fut in concurrent.futures.as_completed(futures):
        result = fut.result()
        print(result, end=" ")
# Возможный вывод: Task2 Task3 Task1 (в порядке завершения задач)

# Используем map:
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    names, durations = zip(*tasks)
    for result in executor.map(sleep_and_return, names, durations):
        print(result, end=" ")
# Вывод: Task1 Task2 Task3 (в порядке как в списке tasks)
```

В этом коде три «задачи» с разным временем работы: Task1 = 3с, Task2 = 1с, Task3 = 2с. При итерации через `as_completed` результаты будут получены в порядке готовности: сначала `Task2` (1 сек), затем `Task3` (2 сек), и затем `Task1` (3 сек). При использовании `map` результаты приходят строго в порядке `Task1, Task2, Task3`, но при этом выполнение *параллельное*: `map` под капотом запускает все задачи сразу. Однако итератор `map` приостанавливается на каждом результате до его готовности. В данном примере он выведет `Task1` только спустя 3 секунды (когда завершится самая долгая первая задача), а затем сразу `Task2` (так как она уже давно готова) и `Task3` (готова к этому моменту).

Когда что применять: - `as_completed()` полезна, когда нужно обрабатывать результаты как можно скорее, в том порядке, как они завершаются (например, обрабатывать ответы от множества веб-запросов по мере поступления). - `Executor.map()` удобна, когда порядок результатов важен, либо код проще написать через map. Но помните, что при `map` самая медленная задача может задержать получение всех остальных результатов.

В обоих случаях, объект `ThreadPoolExecutor` автоматически управляет созданием и завершением потоков. При выходе из блока `with` пул автоматически вызывает `shutdown(wait=True)`, дожидаясь окончания всех задач.

Стоит отметить, что модуль `concurrent.futures` поддерживает не только потоки, но и процессы: класс `ProcessPoolExecutor` аналогичен, но выполняет задачи в отдельных процессах, позволяя обходить ограничения GIL для CPU-нагруженных задач. Однако создание процессов дороже по ресурсам, поэтому для I/O-задач обычно используют пул потоков, а для CPU-задач – пул процессов (или новые возможности, о которых позже).

3. Взаимодействие между потоками: блокировки (Lock) и события (Event)

Когда несколько потоков обращаются к общим данным, возникают *гонки* (race conditions) – ситуации, когда результат зависит от порядка планирования потоков. Чтобы избежать состязания за ресурсы, используются **синхронизаторы**:

- **Lock (замок, mutex)** – механизм взаимного исключения. Только один поток за раз может удерживать замок; другие, пытающиеся его захватить, будут ждать. Это гарантирует эксклюзивный доступ к критической секции кода (обычно работе с разделяемыми данными).

- **Event (событие)** – механизм для организации ожидания условного события. Один или несколько потоков могут ждать (`wait()`) наступления события, а другой поток устанавливает событие (`set()`), после чего все ожидающие потоки разблокируются. События удобны для координации действий потоков (например, не начинать работу, пока не выполнено какое-то условие, или сигнализировать о завершении задачи).

Использование Lock (пример):

Предположим, у нас есть общий счетчик, увеличиваемый из нескольких потоков. Без синхронизации итоговое значение может оказаться неверным из-за гонки (потоки могут одновременно читать старое значение и записывать результат, теряя обновления друг друга). Используем `threading.Lock` для защиты обновления счетчика:

```
import threading

counter = 0
lock = threading.Lock()

def increment(n):
    global counter
    for _ in range(n):
        # критическая секция
        with lock:               # захватываем lock перед изменением
            counter += 1         # безопасно увеличиваем разделяемый ресурс
    # lock автоматически освобождается при выходе из with-блока

# Запустим два потока без блокировки и с блокировкой
```

```
counter = 0
t1 = threading.Thread(target=increment, args=(1000000,))
t2 = threading.Thread(target=increment, args=(1000000,))
t1.start(); t2.start()
t1.join(); t2.join()
print("Итоговый счетчик:", counter)
```

В этом коде два потока суммарно выполняют `increment` 2 миллиона раз. При корректной синхронизации (с `lock`) итог должен быть `2000000`. Без использования `lock` (например, если закомментировать `with lock`) результат мог бы быть меньше из-за потери обновлений. На практике, **GIL в CPython частично упрощает ситуацию** – т.к. исполнение Python-байт-кода потоками происходит по очереди, мелкие операции выглядят атомарными. Однако это обманчиво: даже при наличии GIL некоторые операции не единичны на уровне байт-кода (например, `counter += 1` – это несколько шагов). Если планировщик переключит поток в промежутке между чтением и записью переменной, возникает гонка. Поэтому **лучше всегда явно использовать Lock** для общих данных, чтобы сделать логику независимой от тонкостей планирования и GIL.

Замечание: Объекты многих встроенных типов Python (например, списки, словари) реализованы таким образом, что отдельные операции над ними *атомарны* в контексте GIL. Например, `list.append()` в CPython выполняется под защитой GIL, поэтому два потока не повредят внутрь структуры списка. Но более сложные последовательности операций (например, «проверить и затем изменить») всё равно требуют Lock для корректности.

Использование Event (пример):

Рассмотрим задачу: необходимо запустить несколько потоков-«работников», но они должны начать работу одновременно, *после* того как основной поток даст сигнал (например, дождаться загрузки необходимых данных). Это можно сделать с помощью `threading.Event`:

```
import threading, time

start_event = threading.Event()

def worker(name):
    print(f"{name} ждет сигнала...")
    start_event.wait()           # ждем, пока событие будет установлено
    print(f"{name} начал работу!")

# Создаем и запускаем 3 потока, которые сразу начнут ждать событие
for i in range(3):
    threading.Thread(target=worker, args=(f"Рабочий-{i}",),
daemon=True).start()

# Имитация подготовки в основном потоке
time.sleep(3)
print("Основной поток: готово, даем сигнал начать")
start_event.set()                # устанавливаем событие, разблокируя всех
worker-ов
```

```
# (daemon-потоки завершатся при завершении программы автоматически)
```

Вывод:

```
Рабочий-0 ждет сигнала...
Рабочий-1 ждет сигнала...
Рабочий-2 ждет сигнала...
Основной поток: готово, даем сигнал начать
Рабочий-0 начал работу!
Рабочий-1 начал работу!
Рабочий-2 начал работу!
```

В этом примере три потока заблокированы на `wait()`. Когда основной поток через 3 секунды вызывает `set()`, событие **переходит в сигнальное состояние**, и *все* ожидающие потоки сразу разблокируются практически одновременно. Если бы нужно было после этого вернуть событие в несигнальное состояние (чтобы, например, использовать его повторно), можно вызвать `start_event.clear()`.

Также часто `Event` используется, чтобы **послать сигнал остановки** потокам: например, потоки периодически проверяют `event.is_set()` в цикле, и когда основной поток вызывает `set()`, они завершаются.

Другие синхронизаторы: В модуле `threading` есть и другие примитивы: - `Semaphore` (семафор) – позволяет нескольким потокам одновременно проходить (счетчик ресурсов). - `Condition` (условная переменная) – позволяет более сложное координированное ожидание определённых условий, часто используется вместе с Lock. - `Barrier` – для одновременного синхронного продолжения группы потоков (барьерная синхронизация).

Каждый примитив имеет свои случаи применения, но принципы схожи: обеспечивать правильную координацию доступа к ресурсам или порядка выполнения.

4. Структура `PyThreadState`: роль и связь с GIL

Когда вы запускаете Python-программу, интерпретатор создает внутренние структуры данных для управления выполнением. Одна из ключевых – структура `PyThreadState`, представляющая состояние потока Python на уровне интерпретатора. Проще говоря, `PyThreadState` хранит все, что интерпретатор должен знать о *конкретном потоке* исполнения: текущий выполняющийся frame (стек вызовов Python-функций), текущий код (инструкция байт-кода), указатель на интерпретатор (`PyInterpreterState`), информацию об исключениях, глубину рекурсии и пр.

Каждый поток, выполняющий Python-код, имеет свой объект `PyThreadState`. В CPython существует **глобальный указатель** на текущий `PyThreadState` – тот, который соответствует потоку, владеющему **GIL** сейчас [1] [2]. GIL (Global Interpreter Lock) – глобальная блокировка интерпретатора – защищает этот указатель и другие глобальные структуры.

Связь с GIL: Только один поток может исполнять Python-байт-код в один момент, поэтому обычно только один `PyThreadState` активен (ассоциирован с потоком, который захватил GIL). При

переключении исполнения между потоками, интерпретатор должен: 1. Сохранить текущий `PyThreadState` (состояние текущего потока), 2. Освободить GIL, 3. Захватить GIL потоком-претендентом, 4. Восстановить его `PyThreadState` как текущий.

Эти действия происходят, например, когда один поток освобождает GIL (в ожидании I/O или принудительная уступка), а другой захватывает его, чтобы начать выполняться. В CPython есть функции C-API для этого: `PyEval_SaveThread()` – сохраняет текущий `PyThreadState` и освобождает GIL, возвращая указатель на сохраненное состояние; `PyEval_RestoreThread(tstate)` – захватывает GIL и устанавливает переданный `PyThreadState` как текущий [3]. Эти функции используют в механизмах переключения потоков (о них подробнее в разделе про `time.sleep` ниже).

Структура `PyThreadState` связана один-к-одному с *потоком ОС* (для Python-потоков). **Важно:** один и тот же ОС-поток может последовательно иметь разные `PyThreadState`, если он по очереди исполняет код в разных суб-интерпретаторах (поддерживается C-API). Но одновременно поток ОС работает только с одним `PyThreadState`. В противоположную сторону, один `PyThreadState` *никогда* не используется двумя разными потоками ОС – это строго состояние конкретного потока [4] [5].

В многопоточной программе на Python обычно не приходится напрямую взаимодействовать с `PyThreadState` – все происходит под капотом. Но понимание этой структуры помогает разобраться, как реализованы потоки в CPython. Например, **исключения** хранятся в полях `PyThreadState` – у каждого потока свое текущее исключение, трассировка стека и т.д., благодаря чему исключения не «перемешиваются» между потоками.

Кроме того, `PyThreadState` содержит указатель на `PyInterpreterState` – состояние интерпретатора, к которому принадлежит поток. **Суб-интерпретаторы** (о них далее) имеют свой `PyInterpreterState`, и потоки, запущенные в них, будут иметь поля `interp` указывающие на тот конкретный интерпретатор.

Резюмируя, `PyThreadState` – это структура, которая связывает поток ОС с исполнением Python-кода, а GIL гарантирует, что одновременно модифицировать объекты Python будет только один такой поток, делая работу с `PyThreadState` и другими структурами потокобезопасной внутри интерпретатора.

5. Глобальный интерпретаторный замок (GIL): зачем нужен, как устроен, плюсы и минусы

GIL (Global Interpreter Lock) – это глобальная **блокировка** (mutex), которая **сериализует выполнение байт-кода Python** в CPython. Проще говоря, GIL не позволяет двум потокам одновременно выполнять Python-инструкции. В каждый момент времени только один поток *держит GIL* и выполняет Python-код, остальные либо ожидают GIL, либо находятся в операциях, где GIL временно отпущен.

Зачем введён GIL: Он значительно упрощает реализацию интерпретатора и управление памятью: - **Потокобезопасность внутренних структур:** Без GIL пришлось бы защищать мьютексами большинство операций с объектами Python (увеличение/уменьшение счетчика ссылок, доступ к объектам в памяти, менеджер памяти и т.д.). GIL делает весь интерпретатор по сути однопоточным, избавляя от множества мелких блокировок. В результате **однопоточный**

код работает быстрее, т.к. не тратится время на многочисленные локи при каждой операции [6]. - **Простота интеграции C-расширений:** Многие расширения на C не рассчитаны на работу в многопоточной среде. При наличии GIL можно позволить только одному потоку вызывать C-расширение за раз, и безопасно использовать непотокобезопасные C-библиотеки [7]. - **Проще реализация интерпретатора:** Реализация, где один глобальный лок – гораздо проще, чем тонкая настройка сотен локов или безлоковых структур [8]. Это исторически ускорило разработку CPython.

Как устроен GIL в CPython: GIL реализован как *mutex* (например, pthread mutex на POSIX-системах) плюс некоторый механизм переключения. В старых версиях Python переключение потоков происходило каждые N байт-код инструкций (по умолчанию N=100). В современных версиях (начиная с Python 3.2+) используется **таймер (setswitchinterval)**: поток, исполняющий Python, периодически проверяется (каждые ~5 мс по умолчанию [9]) – если есть другой поток в ожидании GIL, текущий поток отпустит GIL добровольно, позволяя другому потоку выполниться. Этот механизм улучшает公平ность планирования между потоками и реактивность (в старых реализациях были проблемы, когда один поток мог захватить GIL надолго).

Когда поток желает выполнить Python-байт-код, он вызывает `PyEval_AcquireLock()` (внутренне), чтобы захватить GIL. Если GIL занят другим, поток блокируется (ставится в ожидание). При отпускании GIL (например, `PyEval_ReleaseLock()`), другой ожидающий поток пробуждается и захватывает GIL.

Плюсы GIL: - Уже отмеченные *упрощение разработки* и *ускорение однопоточного кода*. GIL убирает накладные расходы на локализацию каждой операции. Однопоточные программы на CPython обычно работают быстрее, чем эквивалент без GIL (потому что нет оверхеда на синхронизацию каждого объекта) [6]. - Большая часть кода CPython и расширений изначально писалась с предположением об эксклюзивности доступа, что снижает количество ошибок, связанных с параллельной памятью. - Неочевидный плюс: GIL позволяет **легко использовать многопоточность для I/O-задач**. Пока один поток ждет сетевой ответ или спит, другой может выполнить Python-код – это работает, потому что многие системные вызовы отпускают GIL (см. следующий раздел). Таким образом, Python-потоки отлично справляются с параллельной обработкой ввода-вывода (файлы, сокеты и т.п.), хотя с CPU-bound задачами все иначе.

Минусы GIL: - **Отсутствие параллелизма на нескольких ядрах для Python-кода.** Если у вас CPU-нагруженная задача и вы запустили несколько потоков, выполнение всё равно будет последовательным (по очереди), и вы не получите ускорения на многопроцессорной машине [9]. Например, вычисление большого массива чисел в 2 потоках не в 2 раза быстрее, а даже чуть медленнее, чем в одном (потери на переключениях и синхронизации). - **Контенция GIL**: Если несколько потоков активно пытаются выполнять код (например, все время что-то считают), они будут постоянно соревноваться за GIL. Это может приводить к эффектам: из-за переключений общий оверхед растет, и на **однопроцессорной системе** параллельный запуск потоков может дать *замедление* относительно строго последовательного выполнения (из-за расходов на ненужные переключения контекста) [9]. - **Блокировка при системных вызовах**: Если расширение или встроенная операция не освобождает GIL на время выполнения долгого системного вызова, то **весь интерпретатор стопорится**. Например, одна плохо реализованная C-функция, которая делает `read()` с диска и не отпустила GIL, задержит все потоки (даже если другим есть что делать). К счастью, почти все длительные операции ввода-вывода в стандартной библиотеке специально освобождают GIL (например, операции с файлами, сокетами, `time.sleep()`, и др. делают это). - **Усложняет использование потоков**: Разработчикам, желающим задействовать несколько ядер, приходится обходить GIL: либо использовать несколько процессов (через `multiprocessing` или внешние управляющие программы), либо C-

расширения (numpy выпускает GIL внутри своих вычислений на C), либо использовать другие интерпретаторы (Jython, IronPython – они без GIL, но со своими нюансами). Само наличие GIL – частый предмет критики, так как не всякую задачу удобно распараллеливать процессами вместо потоков.

Итог: GIL – компромиссное решение: простота и скорость одиночного потока в обмен на отсутствие масштабирования на уровне потоков. Для I/O-bound задач GIL не мешает добиться конкурентности, а для CPU-bound – Python предлагает обойти его с помощью процессов или (в будущем) других средств. Далее мы рассмотрим, как сообщество работает над **free-threading** (свободным от GIL режимом).

6. Как работает `time.sleep(10)` и при чем тут `Py_BEGIN_ALLOW_THREADS` / `Py_END_ALLOW_THREADS`

Функция `time.sleep(secs)` приостанавливает выполнение текущего потока на заданное число секунд, используя системный вызов `sleep` или аналогичный. Если бы при этом GIL оставался захваченным, никакие другие потоки не смогли бы выполняться все эти секунды. Поэтому реализовано иначе: `time.sleep()` **временно освобождает GIL**, позволяя другим потокам работать, пока текущий спит.

В CPython существует пара макросов C API для таких случаев:

- `Py_BEGIN_ALLOW_THREADS` – отмечает начало блока, в котором текущий поток **отпускает GIL**.
- `Py_END_ALLOW_THREADS` – конец такого блока, **возобновление GIL** обратно.

Выглядит это так (упрощенно, на C):

```
Py_BEGIN_ALLOW_THREADS
    // ... здесь выполняется блокирующая операция, не трогающая Python-
объекты ...
    sleep(10);  // системный вызов на 10 секунд
Py_END_ALLOW_THREADS
```

При компиляции эти макросы раскрываются примерно в такой код [10] [11]:

```
PyThreadState *_save;
_save = PyEval_SaveThread();   // освобождаем GIL и сохраняем текущее
состояние потока
sleep(10);                     // выполняем блокирующую операцию (10 секунд)
PyEval_RestoreThread(_save);   // восстанавливаем сохранённый PyThreadState и
GIL
```

Функция `PyEval_SaveThread()` делает две вещи: **освобождает GIL** и сохраняет указатель на текущий `PyThreadState` в переменную (возвращает его). Она должна вызываться, когда поток **уже владел GIL** (иначе отпускать нечего). После этого текущий поток продолжает существовать, но интерпретатор считает, что у него как бы “нет” активного Python-состояния – другими словами, поток спит и не участвует в выполнении Python-кода.

Когда блокирующая операция завершена, вызывается `PyEval_RestoreThread(_save)`. Эта функция **захватывает GIL** заново и устанавливает `PyThreadState` обратно, продолжая исполнение Python-кода с того места. За счёт этого при `time.sleep(10)` (и любом другом вызове, который использует эти макросы) поток не держит GIL и не мешает другим выполняться [12].

Многие операции ввода-вывода в стандартной библиотеке оформлены таким образом: при ожидании данных из сети, чтении файла, задержке, ожидании блокировки и т.п. – GIL отпускается. Например, модули `zlib` и `hashlib` тоже освобождают GIL на время сжатия или хэширования больших буферов [13], что позволяет другим потокам использовать CPU.

С точки зрения Python-разработчика, это прозрачно: вы просто вызываете `time.sleep()`, а на самом деле интерпретатор под капотом сделал нужное, чтобы не блокировать весь процесс. Благодаря этому, как только один поток заснул, другой сразу может получить GIL и выполняться. Аналогично, методы сокетов освобождают GIL, когда ждут данных из сети, поэтому можно параллельно выполнять множество сетевых запросов потоками.

`Py_BEGIN_ALLOW_THREADS` / `Py_END_ALLOW_THREADS` **и их эквиваленты:** Это низкоуровневый API. При написании C-расширений для Python разработчик может явно использовать эти макросы вокруг любой долгой или блокирующей операции, чтобы не держать GIL впустую. Python сам расставляет их во всех ключевых местах стандартной библиотеки. Кроме того, есть связанные утилиты: - `PyGILState_Ensure()` / `PyGILState_Release()` – более высокоуровневый API для работы с GIL, особенно удобный для *потоков, созданных вне Python* (например, если сторонняя C-библиотека запускает свой поток и в нём нужно выполнить Python-код). Эти функции автоматически привязывают `PyThreadState` к текущему потоку ОС и управляют GIL, скрывая прямые вызовы `PyEval_SaveThread` и т.д. [14] [15].

Важно понимать, что `Py_BEGIN_ALLOW_THREADS` нельзя использовать произвольно в Python-коде – это макрос Си для разработчиков интерпретатора/расширений. Но его поведение объясняет, почему, например, `threading.Event.wait()` или `time.sleep()` не мешают другим потокам: внутри они вызывают системное ожидание под защитой этих макросов.

7. Единица планирования в ОС Linux

В операционной системе Linux базовой единицей планирования (scheduling) является **поток исполнения** (thread) – иногда его называют **легковесным процессом** (LWP, light-weight process). С точки зрения планировщика Linux нет большой разницы между “процессом” и “потоком”: и то, и другое – это **task** (задача) с собственным контекстом исполнения (регистры, программный счетчик, стек) [16].

Разница между процессом и потоком в пользовательских терминах в том, что **процессы** не делят память друг с другом (у каждого свой адресный空间), а **потоки внутри одного процесса** разделяют *часть ресурсов* – как правило, общую память (heap, глобальные переменные), открытые файловые дескрипторы и т.д. [17]. В Linux реализация потоков как раз опирается на механизм `clone()`: новый поток – это новая задача (task struct в ядре) с общей памятью с родителем, но имеющая свой **идентификатор (TID)**, свой стек, свои регистры. Ядро планирует эти задачи конкурентно на процессорах.

Проще говоря, **самая маленькая единица, которая может получить квант CPU от планировщика Linux – это поток (task)**. Главный поток процесса тоже является задачей для

планировщика. Если процесс порождает 10 потоков, то для ядра это 10 задач, которые оно по очереди ставит на CPU (или на разные CPU параллельно, если доступно). Таким образом, многопоточность в Linux – это частный случай механизма планирования задач.

С точки зрения Python/CPython: каждый `threading.Thread` – это такой Linux-поток (на Windows – аналогичный Win32 thread), получающий время процессора по решению планировщика ОС. Однако из-за GIL далеко не все эти потоки будут действительно выполнять Python одновременно – GIL становится узким местом в пользовательском пространстве. Тем не менее, ОС не знает о GIL и будет пытаться распределять квант времени каждому потоку справедливо.

Отметим, что **процесс** можно рассматривать как "тяжеловесную" единицу планирования: в контексте некоторых ОС (и в разговорной речи) говорят, что планировщик оперирует процессами. Но в Linux разграничение размыто: процессы и потоки – общие задачи. Поэтому правильнее: *в Linux планируются задачи (threads)*. Например, когда мы запускаем 4 Python-потока, Linux видит 4 задачи и пытается выдавать им процессор по очереди (а внутри CPython уже решает, кто реально поработает через GIL).

8. Стек и куча в контексте многопоточности

Стек (stack) и **куча** (heap) – два разных региона памяти процесса, и многопоточность влияет на работу с ними по-разному:

- **Стек**: Каждый поток получает свой собственный стек. Стек – это область памяти, используемая для хранения контекста вызовов функций, локальных переменных, адресов возврата и т.д. В многопоточной программе *каждый поток имеет отдельный стек*, изолированный от других. Один поток не должен читать/писать стек другого (ядро обычно защищает области стека разными сегментами памяти). Поэтому локальные переменные функций **не разделяются между потоками** – если два потока вызывают одну и ту же функцию, они имеют две независимые копии локальных переменных (на своих стеках). Размер стека потока ограничен (дефолтно в Linux обычно несколько МБ, можно настраивать). Глубокая рекурсия в одном потоке приводит к переполнению его стека (StackOverflow) независимо от других потоков.

- **Куча**: Куча – это область динамической памяти, управляемая аллокатором (например, `malloc` / `free` в C, или внутренний об管理 памяти Python). *Куча общая для всего процесса*. Все потоки в процессе разделяют одну кучу, то есть динамически выделенная память доступна всем потокам. Например, если в одном потоке создать Python-объект (который размещается в куче), другой поток, имея ссылку на этот объект, может к нему обратиться. Это и мощь, и источник проблем: разделяя кучу, потоки могут одновременно пытаться читать/писать одни и те же области памяти, что требует синхронизации.

Python и память: В Python большинство объектов (списки, словари, пользовательские объекты и т.п.) размещаются в куче. Благодаря GIL, внутренний аллокатор CPython (obmalloc) не требует мьютекса при каждом выделении памяти – GIL обеспечивает, что две операции выделения не столкнутся одновременно. Однако в **отсутствие GIL** (см. free-threading) управление кучей должно становиться потокобезопасным, например, через блокировки или другие механизмы.

Влияние на многопоточность: - Поскольку стек у потоков разный, *каждый поток имеет свой контекст выполнения*. Это позволяет потокам выполняться независимо (например, один поток может быть глубоко в рекурсии, а другой – нет, и они не мешают друг другу в этом).

Операционная система при планировании сохраняет/восстанавливает контекст, включая указатель стека, для каждого потока. - Общая куча означает, что **данные, созданные в одном потоке, могут использоваться в другом**, но нужно предусмотреть защиту. Например, два потока добавляют элементы в один список – без GIL пришлось бы использовать Lock вокруг таких операций, иначе структура списка может повредиться. С GIL же в CPython эти операции синхронизированы неявно. - Стоит помнить, что Python объекты распределены по куче, но переменные-ссылки могут жить и на стеке (например, локальные переменные функции – это записи на стеке, указывающие на объекты в куче). Поэтому два потока могут иметь на своих стеках переменные, указывающие на один и тот же объект в куче – и если один поток изменяет объект, другой увидит изменения (если объект изменяемый). Это тоже ситуация, где нужна координация (например, Lock для доступа).

Пример: Пусть у нас есть глобальный список `A = []`. Два потока выполняют функцию, которая делает `A.append(x)` 1000 раз. С GIL каждый `append` по очереди выполнится и в итоге длина списка станет 2000. Без какой-либо блокировки, но при наличии GIL, эта операция пройдёт нормально, потому что реализация списка в CPython делает все изменения под GIL. Но если бы два потока одновременно работали с общей кучей без глобальной блокировки, могло бы произойти пересечение (например, оба потока проверяют размер и размещают элемент в одну и ту же позицию), повредив внутренние данные списка. Поэтому в свободно-потоковой реализации пришлось бы ставить явный mutex на структуру списка.

Резюмируя: **стек изолирован по потокам, куча – разделяемая**. Многопоточность требует защищать операции на куче (разделяемых объектах), но позволяет независимую работу со стеками. GIL в CPython берет на себя большую часть этой защиты, а без GIL ответственность ложится на разработчиков интерпретатора и расширений – или на самого программиста, если он работает на уровне разделяемой памяти.

9. Концепция free-threading и Python

Free-threading – это термин, обозначающий способность интерпретатора исполнять код в нескольких потоках *параллельно*, без глобальной блокировки, т.е. **свободная (полноценная) многопоточность**. В контексте Python, free-threading противоположен наличию GIL.

Исторически, CPython всегда имел GIL, поэтому он *не* free-threaded. Были попытки убрать GIL: - В конце 1990-х Greg Stein создал патч “free-threading” для Python 1.5, который полностью убирал GIL и заменял его множеством мелких блокировок. Однако производительность однопоточного кода с этим патчем падала примерно на 2-3 раза. Патч не был принят в основной проект, показав сложность такой перестройки. - Другие реализации Python: **Jython** (Python на JVM) и **IronPython** (Python на .NET) – *не имеют GIL*. Они полагаются на встроенные возможности JVM и .NET для параллелизма и сборки мусора. Таким образом, Jython и IronPython free-threaded – потоки Java или .NET действительно выполняют Python-код параллельно. Однако эти реализации не совместимы с C-расширениями CPython и имеют свои ограничения. - Проект **PyPy-STM** (Software Transactional Memory) пытался реализовать исполнение Python с помощью транзакционной памяти вместо GIL, но не достиг широкой производственной готовности.

Современные шаги к free-threading в CPython: На данный момент (2025) в сообществе Python реализуется постепенный план по устранению GIL. **PEP 703** "Making the GIL optional" был принят, предусматривая возможность компиляции CPython *без GIL* (`--disable-gil` конфигурация) [18]. Это означает, что в будущем появится режим интерпретатора, где GIL нет, и Python-код действительно может выполняться параллельно на нескольких ядрах.

Однако удаление GIL – сложная задача. Нужно обеспечить потокобезопасность всей инфраструктуры: - В CPython 3.12+ начали появляться изменения в эту сторону: например, введены **"Per-Interpreter GIL"** (отдельный GIL на интерпретатор, PEP 684) и механизмы **изолированных интерпретаторов**. - Добавлены инструменты для безопасного обновления объектов без GIL, например, критические секции (см. следующий раздел). - Переосмыслена работа менеджера памяти (объектного аллокатора). В no-GIL режиме обMalloc отключается общий пул и каждый интерпретатор использует свой (или вовсе поток-специфичные арены), чтобы избегать гонок на уровне `malloc`.

Плюсы free-threading для Python: Очевидный – возможность масштабировать производительность на многоядерных CPU без ухищрений вроде multiprocessing. Например, научные вычисления, веб-серверы, обработка данных – все это могло бы выигрывать прямым использованием потоков.

Минусы и сложности: Удаление GIL может снизить скорость *однопоточных программ* (из-за накладных расходов синхронизации) [19]. Требуется адаптация C-расширений: многие из них не рассчитаны на исполнение без GIL. По этой причине переход планируется *постепенный*: первое время CPython может поддерживать два режима сборки – с GIL (по умолчанию) и без GIL (опционально), чтобы сообщество тестировало совместимость. В будущем, возможно, GIL полностью уберут, если удастся сохранить приемлемую производительность и безопасность.

Связь free-threading с мульти-интерпретаторами: Еще один подход – оставить GIL, но разрешить **несколько интерпретаторов** в процессе, каждому дать свой GIL. Тогда потоки в разных интерпретаторах могут выполняться параллельно. Это реализовано как *"subinterpreters" + "per-interpreter GIL"* (PEP 684). Такой подход сложнее для разработчиков приложений (т.к. объекты не могут делиться между интерпретаторами без явной передачи), но проще обеспечить безопасность и изолированность. Это частично free-threading (между интерпретаторами параллельно, внутри интерпретатора по-прежнему GIL).

В целом, **free-threading Python** – это направление развития, цель которого убрать глобальный замок и позволить настоящую параллельность потоков. Уже в Python 3.13 начали появляться фрагменты этой работы (см. критические секции ниже), а в Python 3.14+ ожидается доступный `InterpreterPoolExecutor` и, возможно, экспериментальный no-GIL режим.

10. Атомарные операции и критические секции в CPython: `Py_BEGIN_CRITICAL_SECTION`

Понятие *атомарности* означает, что операция выполняется как единое целое, без возможности наблюдать ее выполнение частями из параллельных потоков. В многопоточности часто нужно сделать некоторый набор действий атомарно (например, "прочитать значение X, вычислить на его основе Y, записать Y обратно в X" – должно быть непрерываемым, чтобы другой поток не вмешался между чтением и записью). **Критическая секция** – фрагмент кода, который должен выполняться атомарно; обычно достигается с помощью блокировок (mutex).

В CPython с GIL большинство операций над объектами *по факту атомарны* относительно других Python-потоков, потому что GIL не отпустит управление посередине операции на C уровне. Например, увеличение счетчика ссылок объекта, изменение поля объекта, вставка в словарь – все это происходит внутри кода, выполняющегося под GIL, поэтому другие потоки Python не вмешиваются. Разработчики CPython могли не заботиться о блокировках на каждую мелочь – GIL гарантирует целостность.

Однако при переходе к free-threading (без GIL) нужна более тонкая гранулярность защиты. Нельзя больше полагаться на один глобальный замок – потребуется защищать *конкретные объекты* или группы операций. Для этого в Python 3.12+ ввели механизм **«критических секций» на уровне объектов**:

- `Py_BEGIN_CRITICAL_SECTION(obj)` **/** `Py_END_CRITICAL_SECTION()` – эти C-макросы позволяют **захватить встроенный лок объекта** `obj` на время выполнения блока кода [20] [21]. Идея в том, что у каждого объекта Python (который поддерживает такую семантику) есть скрытый mutex, и критическая секция на объекте обеспечивает эксклюзивный доступ к нему. В *обычной сборке CPython с GIL* эти макросы ничего не делают (расширяются в пустые `{}` блоки) [22], т.к. GIL и так защищает все. Но в **free-threaded сборке** эти макросы превращаются в захват/освобождение локов.

- `Py_BEGIN_CRITICAL_SECTION2(obj1, obj2)` / `Py_END_CRITICAL_SECTION2()` – вариация для одновременного захвата двух объектов [23]. Чтобы избежать взаимной блокировки (deadlock), локи берутся в порядке адресов объектов (сравниваются указатели) [23]. Это важно, если нужно атомарно, например, переместить элемент из одного контейнера в другой – надо заблокировать оба.

Пример использования (на уровне C API): Представим метод объекта, который изменяет внутреннее состояние этого объекта. Если во время изменения может произойти что-то, что отпустит GIL (например, вызов пользовательского колбэка, удаление объекта и вызов его финализатора и т.д.), то в no-GIL режиме это опасно – другой поток мог бы залезть в объект в середине изменения. Объявив блок `Py_BEGIN_CRITICAL_SECTION(self); ... Py_END_CRITICAL_SECTION();`, мы гарантируем, что на время изменения **объект** `self` **залочен** и другой поток не войдет в свой метод на том же объекте. Даже если наш код внутри вызовет что-то, что само по себе отпустит GIL (например, ожидание), runtime *приостанавливает действие критической секции*, временно освобождая лок объекта, но помечая, что секция не завершена [24]. Это довольно сложная механика: CPython может *супендировать критическую секцию*, если внутри нее поток все же ждет (чтобы не держать лок слишком долго и не вызвать дедлок). Но при возобновлении, оно гарантирует, что вернет лок обратно и продолжит секцию.

В Python-коде напрямую этих примитивов нет. Но для разработчиков CPython они важны: это инструмент построения **локальных атомарных областей** без GIL. Например, в реализации словаря такие секции могут использоваться, чтобы итерирование по словарю и одновременное изменение не приводили к сбоям – вместо глобального GIL, блокируется конкретный словарь на время итерации [25] [26].

Atomicity vs GIL: Пока GIL существует, разработчики Python-приложений часто не думают об атомарности – они полагаются на GIL. Например, инкремент числа `x += 1` в однопоточном коде – атомарно очевидно; в многопоточно-Python коде – тоже, потому что GIL не даст двум потокам выполнить этот фрагмент одновременно (хотя, как мы обсуждали, в цикле могут быть нюансы). Но в мире без GIL, `x += 1` нужно либо защитить локом на объекте `x` (если x – разделяемый объект), либо использовать атомарную CPU-инструкцию. В C есть атомарные операции (например, атомарный инкремент), и возможно, часть из них будет применена в CPython для счетчиков ссылок или мелких целых. Но для составных структур – нужны критические секции или другие механизмы (lock-free структуры, транзакционная память и пр.).

Также, с точки зрения пользователя, **критическая секция** – это то же самое, что защищенный блок с `threading.Lock`. Если мы говорим о Python-уровне: "начать критическую секцию"

означает *захватить Lock*, "конец" – *освободить Lock*. Так что концепция не новая, просто CPython вводит встроенные локи в объекты, чтобы автоматически использовать их в нужных местах.

В заключение: **атомарные операции** – основа корректности в многопоточности. GIL делал многие операции атомарными глобально. Без GIL, CPython 3.13+ предлагает механизм критических секций на уровне C API, чтобы разработчики могли делать части кода атомарными относительно конкретных объектов. В идеале, для Python-программиста после удаления GIL всё это будет прозрачно – интерпретатор сам будет обеспечивать нужные замки на объекты. Но понимание этого помогает оценить масштаб изменений, необходимых для free-threading.

(Для любознательных: макросы `Py_BEGIN_CRITICAL_SECTION` *и др. были добавлены в Python 3.13* [27] [21]*. В обычной сборке они не влияют на исполнение, а в специальной no-GIL сборке включают логику с* `PyCriticalSection` *структурами.)*

11. *(Опционально)* `InterpreterPoolExecutor` и суб-интерпретаторы: параллелизм с несколькими интерпретаторами

Помимо классических потоков и процессов, в Python развивается идея **суб-интерпретаторов** (subinterpreters). Это отдельные экземпляры интерпретатора Python в пределах одного процесса. Каждый суб-интерпретатор имеет свой `PyInterpreterState` – отдельное пространство: свои модули, свои объекты, свой GIL. Глобальные переменные и модули не разделяются между интерпретаторами [28] [5]. Исторически суб-интерпретаторы были доступны через C API (функция `Py_NewInterpreter`), но не использовались широко.

Почему суб-интерпретаторы интересны? Потому что если каждый интерпретатор имеет свой GIL (изолирован), то можно в одном процессе запускать несколько интерпретаторов *параллельно* на разных потоках ОС. Таким образом, достигается параллелизм, но без риска общей гонки данных, так как объекты из разных интерпретаторов *полностью изолированы* друг от друга (как в отдельных процессах, но легче создавать и без полного разделения памяти на уровне ОС).

В новых версиях Python (PEP 554, PEP 684, PEP 704 и др.) ведётся работа по **внедрению суб-интерпретаторов в стандартную библиотеку**. Ожидается модуль `interpreters` (в некоторых версиях Python 3.12+ он может быть экспериментальным), позволяющий создавать и управлять интерпретаторами из Python-кода. С этим же связан класс `InterpreterPoolExecutor` – аналог `ThreadPoolExecutor`, запускающий задачи на пуле суб-интерпретаторов.

Как это работает: - При создании нового интерпретатора (например, `interp = interpreters.create()`) вы получаете объект, представляющий суб-интерпретатор. Вы можете запустить в нём код с помощью метода `interp.run(source)` или выполнить функцию. - У суб-интерпретатора свой набор модулей, своё пространство имен. По умолчанию он *не имеет доступа* к объектам главного интерпретатора (за исключением низкоуровневых вещей вроде файловых дескрипторов ОС). - Для обмена данными предлагается использовать специальные **каналы (channel, queue)** – например, `interpreters.create_queue()` создаёт очередь, которую можно передать интерпретатору для общения [29] [30]. Либо данные можно сериализовать (pickle) при передаче.

InterpreterPoolExecutor: Это класс в `concurrent.futures`, предложенный (PEP 554/PEP 734) для удобного использования суб-интерпретаторов. Он расширяет `ThreadPoolExecutor`, но

каждый поток пула имеет *свой интерпретатор* [31]. По сути: - При создании пула `InterpreterPoolExecutor(max_workers=N)`, он создаст N потоков ОС, и в каждом — новый суб-интерпретатор. - Когда вы сабмитите задачу (например, через `executor.submit(func, *args)` или `executor.map(func, iterable)`), задача будет отправлена в один из суб-интерпретаторов для исполнения. - Аргументы и возвращаемые значения будут **сериализованы (например, с помощью pickle)** при передаче между главным интерпретатором и рабочим [32]. Это похоже на то, как работает `ProcessPoolExecutor` (он тоже сериализует данные между процессами). Ограничение необходимо, ведь напрямую объекты нельзя шарить между изолированными интерпретаторами. - Зато CPU-bound задачи действительно будут выполняться параллельно, поскольку разные интерпретаторы не блокируют друг друга общим GIL.

Пример использования (условный, API может отличаться в конкретной версии):

```
from concurrent.futures import InterpreterPoolExecutor

def heavy_computation(x):
    # какая-то CPU-интенсивная работа
    total = 0
    for i in range(10**7):
        total += (x * i) % 12345
    return total

with InterpreterPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(heavy_computation, [10, 20, 30, 40]))
print(results)
```

Здесь 4 задачи выполнятся параллельно в 4 суб-интерпретаторах (каждый на своем потоке). Даже если `heavy_computation` целиком на Python, все 4 ядра будут заняты – это даст почти линейный выигрыш, чего не было бы с обычным `ThreadPoolExecutor` из-за GIL.

По состоянию на Python 3.13, `InterpreterPoolExecutor` находится в стадии внедрения. Согласно PEP 734, планируется его появление в Python 3.14 (возможно, и backport в 3.13) [33]. Это значит, что в ближайших версиях Python у нас появится штатное средство для параллелизма, сочетающее удобство потоков (общий адресный пространство, низкие накладные расходы на создание) с отсутствием GIL-конфликта, как у процессов.

Ограничения суб-интерпретаторов: Не все модули готовы работать в нескольких интерпретаторах. Некоторые расширения, хранящие глобальное состояние, могут вести себя неправильно. В CPython вводится флаг `sys.implementation.supports_isolated_interpreters`, указывающий, поддерживает ли данная сборка режим изолированных интерпретаторов [34]. В идеале, со временем все стандартные модули будут безопасны для суб-интерпретаторов. Кроме того, существуют ограничения на объекты, которые можно передавать между интерпретаторами: обычно только данные, сериализуемые `pickle`, или специально поддерживаемые "канальные объекты".

Когда применять: Subinterpreters и InterpreterPoolExecutor могут быть полезны, когда нужно параллельно выполнять вычисления в рамках одной программы без запуска дополнительных процессов. Например, обработка нескольких запросов в веб-сервере, вычисления в науке (если

не хочется использовать multiprocessing из-за overhead), или безопасное параллельное выполнение плагинов/скриптов (каждый в своей песочнице).

В заключение, эта функциональность – шаг навстречу миру без GIL: даже если глобальный GIL остался, мы можем эффективно обойти его, распределив нагрузку по независимым интерпретаторам внутри процесса. Это сложнее, чем традиционные потоки (из-за передачи данных через очереди), но проще, чем работать с отдельными процессами (нет полного раздвоения памяти и необходимости межпроцессного взаимодействия через ОС).

Суммируя всё вышесказанное: Python предоставляет богатый инструментарий для многопоточности и параллелизма, но классический CPython ограничен **GIL**, что влияет на дизайн многопоточных программ. Мы рассмотрели практические аспекты (работа с `threading` и `concurrent.futures`, синхронизация потоков), а также заглянули под капот CPython: как устроено состояние потоков (`PyThreadState`), почему существует GIL и как он реализован, и какие идут изменения (free-threading, суб-интерпретаторы) в новейших версиях Python. Понимание этих тем позволяет писать эффективный и безопасный код, а также готовиться к будущему Python без GIL, где параллельное программирование станет еще более мощным инструментом.

---

1 2 3 10 11 12 13 14 15 20 21 22 23 24 25 27 Initialization, Finalization, and Threads — Python 3.13.3 documentation
https://docs.python.org/3/c-api/init.html

4 5 28 29 30 31 32 34 PEP 734 – Multiple Interpreters in the Stdlib | peps.python.org
https://peps.python.org/pep-0734/

6 7 8 9 Global interpreter lock - Wikipedia
https://en.wikipedia.org/wiki/Global_interpreter_lock

16 17 Поток выполнения — Википедия
https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F

18 19 PEP 703 – Making the Global Interpreter Lock Optional in CPython | peps.python.org
https://peps.python.org/pep-0703/

26 Dictionary Objects — Python 3.13.3 documentation
https://docs.python.org/3/c-api/dict.html

33 Support for SubInterpreters and InterpreterPoolExecutors · Issue ...
https://github.com/apache/arrow/issues/44511