ChatGPT

Многопроцессность и многопоточность в Python

В этом документе рассматриваются механизмы параллелизма в Python на уровне процессов и потоков. Мы подробно разберем создание и управление процессами через модуль `multiprocessing`, использование пула потоков и процессов через `concurrent.futures`, методы межпроцессного взаимодействия (очереди, каналы, сокеты), а также затронем вопросы организации памяти процесса, разделения пространств пользователя и ядра, отличие процессов от потоков (в том числе роль TLB и GIL), и сравним плюсы и минусы многопоточности и многопроцессности.

1. Работа с `multiprocessing.Process`

Модуль `multiprocessing` в Python позволяет запускать функции в отдельных процессах, предоставляя API, похожий на модуль `threading`. Главное преимущество – процессы имеют свой собственный интерпретатор Python и память, что обходит ограничение GIL и позволяет использовать несколько CPU [1]. Класс `multiprocessing.Process` представляет отдельный процесс. Ниже показан пример запуска нескольких процессов с помощью этого класса:

```
from multiprocessing import Process
import os, time

def worker(num):
    print(f"Процесс {num} (PID={os.getpid()}) начал работу")
    time.sleep(1)
    print(f"Процесс {num} завершился")

if __name__ == "__main__":
    processes = []
    for i in range(3):
        p = Process(target=worker, args=(i,))
        processes.append(p)
        p.start()             # запускаем процесс

    for p in processes:
        p.join()              # ожидаем завершения процесса

    print("Главный процесс: все дочерние процессы завершены")
```

В этом примере создаются три объекта `Process` с целевой функцией `worker`. Вызов `p.start()` породит новый дочерний процесс, который начнёт выполнение функции `worker`. Метод `start()` **создает новый процесс на уровне ОС** – в Unix это обычно происходит через системный вызов `fork` (или `posix_spawn` / `vfork` в зависимости от настроек), а в Windows – через вызов `CreateProcess`, запускающий новый экземпляр интерпретатора Python. При

вызове `start()` родительский процесс продолжает свое выполнение, а дочерний процесс выполняет указанную функцию.

Метод `p.join()` используется для синхронизации – он приостанавливает (блокирует) выполнение текущего (родительского) процесса до тех пор, пока соответствующий дочерний процесс не завершится [2]. Проще говоря, `join()` позволяет "дождаться" окончания работы процесса. Если не вызвать `join()`, главный процесс может завершиться раньше дочерних; однако по умолчанию, **не-деамонические** дочерние процессы в Python будут автоматически дожданы при завершении главного процесса [3]. На уровне ОС вызов `join()` обёртывает системный вызов ожидания (например, `waitpid`), который получает статус завершения дочернего процесса и освобождает связанные с ним ресурсы.

Шаги выполнения при вызове `Process.start()` **(Unix)**: (1) Главный процесс вызывает `fork()`, создавая копию себя. (2) В дочернем процессе `multiprocessing` переинициализирует среду: сбрасывает дескрипторы, настраивает связь с родителем и начинает выполнение функции `target` в новом процессе. (3) Метод `start()` возвращается в родительском процессе, а дочерний выполняет задачу. **На Windows** отсутствует `fork()`, поэтому (1) запускается новый процесс через `CreateProcess`, (2) в новом процессе импортируется главный модуль и выполняется целевая функция. Важно учитывать, что на Windows и macOS (начиная с Python 3.8) по умолчанию используется метод запуска *spawn* (создание нового процесса), тогда как на Linux часто используется *fork* [4]. При *fork*-запуске дочерний процесс наследует адресное пространство родителя (память копируется по необходимости, механизм *copy-on-write*), что быстрее, но может приводить к проблемам при наличии потоков. При *spawn*-запуске каждый процесс стартует с чистым интерпретатором, импортируя заново код (более надежно, но чуть медленнее) [5]. В обоих случаях новый процесс получает свой уникальный PID и полностью изолированную память.

Метод `Process.join()` **внутри себя** проверяет завершился ли процесс, и при необходимости вызывает системное ожидание. Если задать `p.join(timeout)`, можно ограничить время ожидания. Также у процесса есть флаг `daemon`. Если установить `p.daemon = True` перед `start()`, то процесс станет *демоном* – он будет автоматически убит при завершении главного процесса, и `join()` для него вызывать не требуется (наоборот, вызов `join` для демона вызовет ошибку). Демоны полезны для фоновых задач, но важно помнить, что при их принудительном завершении блокировки и открытые ресурсы могут не освободиться.

Что происходит на уровне ОС при завершении процесса? Когда функция в дочернем процессе отработала, процесс завершается (через `exit`), ядро ОС помечает его как *зомби* до тех пор, пока родитель не прочитает статус завершения (это делает `join`). После этого запись о процессе удаляется из таблицы процессов. Таким образом, `join()` не только позволяет синхронизироваться, но и предотвращает накопление зомби-процессов.

2. Работа с `concurrent.futures`: `ThreadPoolExecutor` и `ProcessPoolExecutor`

Модуль `concurrent.futures` предоставляет высокоуровневый интерфейс для параллельного выполнения задач с помощью пула исполнителей. Он поддерживает два типа пулов: потоковый (`ThreadPoolExecutor`) и процессный (`ProcessPoolExecutor`), реализующие общий интерфейс `Executor` [6]. Использование пула упрощает управление большим числом задач, автоматически создавая нужное количество потоков или процессов.

ThreadPoolExecutor vs ProcessPoolExecutor: Первый выполняет задачи в разных потоках внутри одного процесса, второй – в отдельных дочерних процессах. Оба имеют схожие методы, такие как `submit`, `map`, `as_completed` и т.д. Выбор между ними зависит от характера задач:
- `ThreadPoolExecutor` хорошо подходит для I/O-bound задач (сетевые запросы, работа с файлами и т.п.), где потоки могут параллельно ждать операций ввода-вывода. Однако для CPU-bound задач эффективность потоков ограничена GIL (в каждом процессе Python одновременно исполняется лишь один поток Python-кода).
- `ProcessPoolExecutor` запускает несколько процессов Python, обрабатывая задачи параллельно на разных ядрах CPU, что эффективно для вычислительно интенсивных задач, обходя GIL [1]. Но обмен данными между процессами требует сериализации (pickle) и межпроцессного взаимодействия, что дороже, чем обмен между потоками (общая память).

Метод `Executor.map` vs функция `as_completed`:
- `Executor.map(func, *iterables)` запускает функцию `func` для каждого элемента из *iterables* и возвращает итератор результатов **в том же порядке, что и входные данные** [7]. То есть, даже если какие-то задачи завершились раньше других, `map` будет ждать результата первого задания, затем второго и т.д. Таким образом, порядок выдачи результатов сохраняется, но это может приводить к неэффективному проставлению – быстрые задачи могут ждать, пока завершатся более медленные, если те идут раньше. Кроме того, если одна из задач возбуждает исключение, это исключение пробрасывается при получении соответствующего результата через итератор `map`, и дальнейшие результаты не будут получены (фактически выполнение `map` прерывается на первом исключении) [8] [9]. Поэтому при использовании `map` следует либо обрабатывать исключения внутри функций, либо быть готовым к тому, что первое же непойманное исключение остановит всю последовательность.
- `concurrent.futures.as_completed(futures)` принимает список объектов `Future` (например, возвращенных из `submit`) и возвращает генератор, порождающий `Future` по мере завершения задач. Иными словами, с помощью `as_completed` можно итерироваться по результатам **в порядке готовности задач**, вне зависимости от исходного порядка запуска [7]. Это удобно, когда нужно сразу обрабатывать результаты по мере их поступления. Пример использования:

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import math, time

def work(x):
    time.sleep(x)  # имитируем разное время выполнения
    return math.factorial(x)

with ThreadPoolExecutor(max_workers=4) as executor:
    futures = [executor.submit(work, i) for i in [5, 3, 4, 2]]
    for future in as_completed(futures):
        result = future.result()
        print("Результат:", result)
```

Здесь задачи (вычисление факториала с задержкой) будут завершаться в порядке [2, 3, 4, 5] секунд, и благодаря `as_completed` результаты тоже выведутся в порядке готовности, а не в порядке отправки. Если же использовать `executor.map(work, [5,3,4,2])` и проитерироваться по результатам, то вывод пойдёт в порядке 5,3,4,2 – при этом выполнение первой задачи (5 секунд) задержит получение результатов остальных, даже если они уже готовы.

Блокировки и поведение: Важно понимать, что **и потоки, и процессы выполняют задания параллельно**, разница лишь в способе получения результатов. Метод `map` **блокирует** ход программы при попытке итерироваться по результатам, если следующий результат еще не готов (ожидая его), тогда как `as_completed` позволяет асинхронно обрабатывать завершенные задачи, не дожидаясь остальных. При этом оба подхода можно сделать неблокирующими, если проверять состояние `Future` или задавать таймауты. Также есть метод `wait()` для ожидания группы задач.

Пример с `ProcessPoolExecutor`: Работа аналогична ThreadPool, но под капотом задачи отправляются в форкнутые процессы. Например:

```
from concurrent.futures import ProcessPoolExecutor
def heavy(x):
    # некая CPU-нагрузка
    return sum(i*i for i in range(x))

with ProcessPoolExecutor(max_workers=4) as executor:
    results = executor.map(heavy, [10000000, 20000000, 30000000, 40000000])
    for res in results:
        print(res)
```

В этом примере четыре больших расчета распределятся между четырьмя процессами и выполнятся параллельно на нескольких ядрах. Без использования процессов они выполнялись бы последовательно значительно дольше.

При работе с `ProcessPoolExecutor` следует помнить, что передача аргументов и возвращаемых значений функций между процессами происходит через сериализацию объектов (модуль `pickle`). Это накладные расходы: большие объёмы данных лучше не гонять через `ProcessPool`, или же использовать разделяемую память (например, `multiprocessing.Array` или новую функциональность `shared_memory`). В `ThreadPoolExecutor` такие данные передаются по ссылке внутри процесса, что быстрее. Также некоторые объекты, не поддерживаемые для pickle, нельзя напрямую передать в дочерний процесс.

Итого: `ThreadPoolExecutor` удобен для параллельного выполнения множества операций ввода-вывода, `ProcessPoolExecutor` – для параллельных вычислений. Комбинация `submit` + `as_completed` дает максимальную гибкость в обработке результатов. Не забывайте корректно закрывать пулы (контекстный менеджер `with` делает это автоматически) – при выходе он дождётся окончания всех задач и завершит потоки/процессы пула.

3. Межпроцессное взаимодействие: очереди, каналы, сокеты

Когда используется `multiprocessing` или просто несколько независимых процессов, возникает необходимость обмениваться данными между процессами. Поскольку у разных процессов *свои отдельные адресные пространства*, они не могут обращаться напрямую к переменным друг друга. Для обмена используются механизмы IPC (Inter-Process Communication). В Python основные варианты: **очереди (Queue)**, **каналы (Pipe)** и **сокеты (Socket)**. Рассмотрим их и сценарии применения.

Очереди (Queue)

Класс `multiprocessing.Queue` предоставляет межпроцессную очередь сообщений (очередь FIFO), похожую на обычную `queue.Queue`, но безопасную для использования между процессами [10] [11]. Один процесс может помещать объекты в очередь через `put()`, другой – получать через `get()`. Пример:

```
from multiprocessing import Process, Queue

def producer(q):
    for item in ["A", "B", "C"]:
        q.put(item)
    q.put(None)  # маркер завершения

def consumer(q):
    while True:
        data = q.get()
        if data is None:
            break
        print(f"Получено: {data}")

if __name__ == "__main__":
    q = Queue()
    p1 = Process(target=producer, args=(q,))
    p2 = Process(target=consumer, args=(q,))
    p1.start(); p2.start()
    p1.join(); p2.join()
```

Здесь один процесс-«производитель» кладет строки в очередь, другой – «потребитель» – извлекает их и печатает. Очередь безопасна для нескольких писателей и читателей, она самостоятельно осуществляет нужную блокировку. **Внутри реализации** `multiprocessing.Queue` использует канал `Pipe` и фоновые потоки для передачи данных между процессами, обеспечивая потокобезопасность. Каждый объект, помещаемый в очередь, **сериализуется** с помощью `pickle` [11], а при чтении – десериализуется обратно. Поэтому через очередь можно передавать *любые pickle-совместимые объекты*, но нужно учитывать накладные расходы на сериализацию.

Очереди удобны, когда нужно передавать задания и собирать результаты от множества процессов – например, пул воркеров может брать задачи из очереди. Класс `JoinableQueue` – вариант очереди, к которой можно вызывать `task_done()` и `join()` для ожидания обработки всех задач.

Когда использовать очередь: когда необходим обмен сообщениями в режиме «многие-ко-многим» (несколько источников и/или получателей). В отличие от канала, очередь может иметь больше двух концов и автоматически управляет конкурентным доступом [12]. Однако за универсальность приходится платить скоростью – очереди чуть медленнее каналов, так как построены поверх них и имеют дополнительный слой управления [13].

Каналы (Pipe)

Функция `multiprocessing.Pipe()` предоставляет низкоуровневый двунаправленный канал между двумя процессами. Она возвращает пару подключений `conn1` и `conn2`, которые представляют два конца канала [14] [15]. Каждый конец имеет методы `send(obj)` и `recv()`. Например:

```
from multiprocessing import Process, Pipe

def sender(conn):
    for x in [1, 2, 3]:
        conn.send(x)
        print(f"[sender] отправил {x}")
    conn.send(None)  # сигнал конца
    conn.close()

def receiver(conn):
    while True:
        data = conn.recv()
        if data is None:
            break
        print(f"[receiver] получил {data}")

if __name__ == "__main__":
    parent_conn, child_conn = Pipe()
    p1 = Process(target=sender, args=(parent_conn,))
    p2 = Process(target=receiver, args=(child_conn,))
    p1.start(); p2.start()
    p1.join(); p2.join()
```

В этом примере один процесс через `conn.send` отправляет числа, другой через `conn.recv` получает. Отправка и получение блокирующие (если нужно, можно использовать `conn.poll()` для проверки, или задать таймаут). **Канал имеет всего два конца** – это *точка-точка соединение* [12]. По умолчанию Pipe — двунаправленный (duplex=True), то есть каждый конец может и посылать, и принимать. При желании можно создать односторонний канал `Pipe(duplex=False)`, тогда один конец только отправляет, а второй только читает.

Как и очередь, `Pipe` при отправке объекта выполняет его сериализацию, а при чтении — десериализацию [16]. Каналы обычно строятся на механизмах ОС: в Unix используются анонимные каналы (pipe) или сокет-пары, в Windows – именованные каналы. **Когда использовать Pipe:** когда нужно организовать двусторонний обмен данными именно между двумя процессами (например, между родительским и одним дочерним). Каналы эффективнее очередей (меньше оверхеда), но менее гибкие – если требуется подключить больше участников, придется либо организовывать несколько Pipe, либо использовать очередь.

Сравнение Queue vs Pipe: Если общение требуется только между двумя фиксированными процессами, `Pipe` будет более быстрым и простым вариантом [17]. Если же нужно фан-аути (например, один процесс рассылает сообщения многим) или сбор от многих источников – лучше использовать `Queue`, которая уже реализует этот функционал. При большом объеме данных канал может переполняться: у Pipe есть ограниченный буфер в памяти ядра (например, 64 КБ),

при заполнении `send()` заблокируется до тех пор, пока другой конец не прочитает часть данных [18] (это регулирование потока).

Сокеты (Socket)

Сокеты – наиболее универсальный способ межпроцессного взаимодействия, позволяющий обмениваться данными как между процессами на одной машине, так и по сети между разными узлами. В контексте Python сокеты предоставляются модулем `socket`. Сокет можно представить как конечную точку соединения, привязанную к определенному адресу (например, IP + порт для TCP/IP). Два процесса могут обмениваться данными, если один из них выступает сервером (слушает адрес), а другой – клиентом (подключается к серверу).

Пример (локально через TCP): один процесс создаст слушающий сокет, второй подключится:

```
# server.py
import socket
srv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv_sock.bind(("localhost", 5000))
srv_sock.listen()
conn, addr = srv_sock.accept()
data = conn.recv(1024)
print("Сервер получил:", data.decode())
conn.send(b"pong")
conn.close()
srv_sock.close()
```

```
# client.py
import socket
cli_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cli_sock.connect(("localhost", 5000))
cli_sock.send(b"ping")
response = cli_sock.recv(1024)
print("Клиент получил:", response.decode())
cli_sock.close()
```

Запустив `server.py` и `client.py` в отдельных процессах (или на разных машинах), мы увидим обмен сообщениями "ping" -> "pong". Здесь использован TCP-сокет. Для общения на одной машине без сети можно использовать UNIX domain socket (AF_UNIX) – тогда адресом будет путь файлового сокета.

Когда использовать сокеты: Когда процессы не имеют общего родителя или должны работать распределенно. Например, микросервисная архитектура или взаимодействие между разными приложениями используют сокеты (TCP/UDP). Внутри одной машины, если нужно общение не только между родителем/потомком, а более гибкое (или языки программирования разные), тоже используют сокеты. Однако сокеты сложнее в использовании: нужно управлять подключениями, обработкой байтовых потоков, протоколами. В Python можно упростить, используя библиотеки (например, `multiprocessing.connection` предоставляет оболочку над сокетами/каналами,

позволяя легко отправлять объекты между процессами, даже удаленными, через `Listener` и `Client`).

Безопасность доступа: Ни очереди, ни каналы, ни сокеты не позволяют двум процессам одновременно использовать одну и ту же переменную в памяти без явного на то механизма. Если нужно разделяемая память, `multiprocessing` предлагает класс `Value`, `Array` или разделяемые блоки памяти (`SharedMemory`). Но в большинстве случаев проще организовать передачу копий данных через очереди/каналы, чем заморачиваться с разделяемой памятью и синхронизацией.

Итог:
- *Очередь* (Queue) – высокоуровневый механизм "публикации-подписки" для нескольких писателей/читателей, удобный и безопасный, но с накладными расходами (данные сериализуются и проходят через промежуточный буфер).
- *Канал* (Pipe) – легковесное решение для двусторонней связи *точка-точка* между двумя процессами, более производительное, но ограниченное двумя участниками.
- *Сокет* (Socket) – универсальный инструмент IPC, особенно для раздельных или распределенных процессов; требует ручной организации протокола обмена, но позволяет коммуникацию в любых топологиях (один-ко-многим, многие-ко-многим) и между машинами.

Выбор зависит от сценария: для простого случая родитель-дочерний процесс часто достаточно Pipe; для пула воркеров – Queue; для разных программ – сокеты.

4. Отображение памяти процесса и управление памятью

Каждый процесс в современной ОС имеет свое виртуальное адресное пространство. Виртуальная память позволяет изолировать процессы друг от друга и абстрагировать адреса от физической оперативной памяти [19]. Рассмотрим, как организована память процесса, как ее просматривать и что такое страницы, таблицы страниц, TLB, heap/stack, *lazy allocation* и др.

Просмотр памяти процесса: `top`, `ps`, `pmap` и др.

Существует несколько утилит для мониторинга потребления памяти процессом: - `top` – утилита реального времени, показывающая процессы и их ресурсы. В колонках `VIRT` (virtual size) и `RES` (resident set) можно увидеть соответственно размер виртуальной памяти процесса и объем физически резидентной памяти (в RAM). Например, `VIRT` включает суммарно весь адресный 공간 (включая загруженные библиотеки, резерв под стек, кучу, и неиспользуемые зарезервированные области), а `RES` – только текущие загруженные в RAM страницы (не включая выгруженные на диск или разделяемые с другими процессами). - `ps` – единовременный снимок процессов. С опцией `-o` можно вывести интересующие поля, например `ps -o pid,vsz,rss,comm -p <PID>` покажет виртуальный (VSZ) и резидентный (RSS) размеры памяти процесса с PID. - `pmap` – утилита, показывающая карту памяти процесса: список всех участков адресного пространства с их адресами, размерами, правами и описанием (например, какой файл отображен или `[heap]`, `[stack]` и т.д.). Пример вывода `pmap`:

```
$ pmap 12345
12345: python myscript.py
0000555555551000   132K r-x-- python3.9
0000555555573000     8K r---- python3.9
```

```
0000555555575000     4K rw--- python3.9
0000555555576000   256K rw---   [ anon ]       # heap может начинаться тут
...
00007ffff7ffa000     4K r----   [ anon ]
00007ffff7ffb000   132K rw---   [ stack ]      # стек основного потока
...
```

Здесь мы видим сегмент кода Python, участок анонимной памяти (вероятно, куча), стек и пр. Метки `[heap]` и `[stack]` позволяют быстро найти расположение кучи и стека в адресном пространстве [20].

Используя `pmap`, можно оценить, сколько памяти занимает сам код, библиотеки, какая часть занята под кучу (динамические аллокации), размер стека и т.д. Также есть специальные файлы в Linux: `/proc/<PID>/maps` (похоже на `pmap`), `/proc/<PID>/smaps` (более подробная статистика по каждому сегменту, включая RSS, грязные страницы и пр.), `/proc/<PID>/status` (сводная информация, включая VmSize, VmRSS и др.).

Виртуальная память, страницы и таблицы страниц

Виртуальная память – это абстракция, предоставляющая каждому процессу иллюзию собственной непрерывной памяти, начинающейся с адреса 0. На самом деле физическая память (RAM) разделяется на *страницы* фиксированного размера (обычно 4 КБ). **Страница** – минимальная единица управления памятью. **Таблица страниц** – структура данных, с помощью которой ОС хранит отображение (mapping) виртуальных страниц процесса в физические фреймы памяти [21]. Проще говоря, таблица страниц – это своего рода справочник: для каждой виртуальной страницы хранится номер соответствующей физической страницы (а также флаги: присутствует ли в памяти, права доступа, была ли модифицирована и пр.).

Когда процесс обращается по какому-либо виртуальному адресу, **MMU (Memory Management Unit)** процессора автоматически переводит его в физический адрес, глядя в таблицу страниц текущего процесса. При этом используются два регистра: регистр базы таблицы страниц (например, `CR3` на x86), указывающий на корень структуры страниц для активного процесса, и специальные специализированные кеши, о них ниже. Современные системы используют многоуровневые таблицы страниц (например, 4-уровневые на x86_64 [22]), разбивая адрес на несколько частей (директории/поддиректории страниц), чтобы не хранить один гигантский массив для всего адресного пространства.

Если требуемая страница помечена как отсутствующая в RAM (например, она еще не выделена или выгружена в swap), возникает исключение *page fault*. Ядро ОС обрабатывает его: либо выделяет физическую страницу (в случае *demand zero* или *lazy allocation*, когда мы впервые обратились к адресу, для которого еще не было физической памяти), либо подгружает страницу с диска (в случае swap или memory-mapped файла), затем обновляет таблицу страниц и продолжает выполнение процесса.

Защита памяти: Таблицы страниц также обеспечивают защиту – одна программа не может обратиться к памяти другой, потому что ее таблица страниц не содержит соответствующих отображений. Попытка доступа к чужому адресу приведет к нарушению защиты (segmentation fault). Также в записях таблицы страниц есть биты прав (на чтение, запись, исполнение), и попытка, например, записи в страницу кода (помеченную только на чтение/исполнение) вызовет fault.

TLB – буфер трансляции адресов: Обращение к таблице страниц – относительно долгий процесс (несколько обращений к памяти для многоуровневой таблицы). Поэтому процессоры используют специальный кеш последних преобразований – Translation Lookaside Buffer. **TLB** хранит небольшое количество недавно использованных соответствий "виртуальная страница -> физическая страница". При каждом обращении к памяти аппаратно сначала проверяется TLB: если там есть запись для нужной страницы (TLB hit), сразу получаем физический адрес [23]. Если же в TLB нет записи (TLB miss), то производится стандартный поиск по таблице страниц (может занимать десятки тактов), а затем результат помещается в TLB [24]. Эффективность TLB существенно влияет на производительность памяти. Размер TLB ограничен (например, у CPU может быть TLB на 64 или 128 записей для данных и инструкций отдельно). Когда процесс выполняется, TLB наполняется его записями. При переключении процессов (контекстный переключатель) обычно происходит очистка или маркировка TLB, т.к. новый процесс имеет свою таблицу страниц (хотя на современных CPU есть расширения – например, ASID, Process Context ID – позволяющие хранить записи TLB для разных процессов одновременно, помечая их принадлежность, чтобы не очищать полностью).

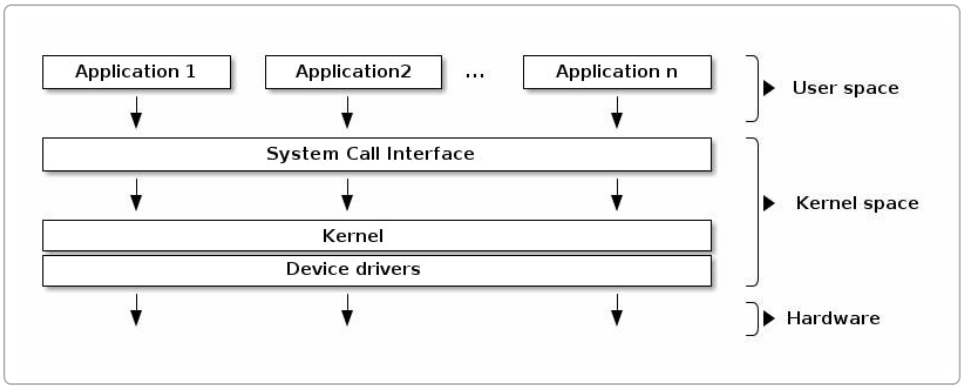
Узнать размеры TLB можно из технической документации CPU. Операционная система не предоставляет их явно через стандартные утилиты, однако в `/proc/cpuinfo` на некоторых архитектурах (например, MIPS) можно увидеть поля о TLB. Например, для процессора Loongson вывод содержит строку `TLB entries: 2112` [25]. В общем случае для x86 можно найти информацию в спецификации или через утилиты типа `lscpu` (которая, впрочем, тоже обычно не показывает TLB). Непрямой путь – запуск специальных бенчмарков, выявляющих размер TLB по скачкам времени доступа, однако в повседневной разработке это не требуется.

Куча и стек: организация и рост

Стек (stack) – область памяти, используемая для хранения локальных переменных, возвращаемых адресов функций, контекста выполнения. В каждом процессе (и потоке) есть свой стек. У основного потока процесса выделяется начальный стек фиксированного размера (например, 8 МБ по умолчанию в Linux). **Стек растёт «вниз»** – то есть от высоких адресов к низким [26]. В адресном пространстве процесса обычно стек находится в верхней части (ближе к максимальным адресам). При каждом вызове функции в стек помещается новый фрейм (структура с локальными переменными, параметрами, адресом возврата), при выходе – убирается. Если стек заполнится (например, бесконечной рекурсией), случится *Stack Overflow* – попытка задействовать память за пределами стека, что обычно приводит к аварийному завершению процесса.

Куча (heap) – область динамической памяти, из которой выделяются блоки по запросу через функции вроде `malloc` (в C) или автоматически под объекты (в Python через malloc внутри интерпретатора). **Куча растёт «вверх»** – от низких адресов к высоким [26]. Как правило, куча располагается сразу после сегментов приложения (кода и статических данных) и до зоны, отведенной под стек. В простейшем случае, если куче нужно больше памяти, она запрашивает у ОС расширение – через системный вызов `brk` / `sbrk` сдвигается «граница кучи» вверх, выделяя новый участок виртуальной памяти для кучи. Если же куча и стек сблизятся вплотную – свободная память исчерпана (в современных системах, однако, адресное пространство настолько велико, что обычно между кучей и стеком оставляется большой «пустой зазор»). Помимо `brk`, большие аллокации могут запрашиваться через `mmap` – выделяется отдельный участок памяти (например, Python для больших объектов может вызывать `mmap` вместо увеличения brk).

Диаграмма: расположение основных сегментов памяти процесса. Ниже схематично показано адресное пространство:



Пространство памяти процесса: внизу – сегменты кода/данных, над ними куча (растет вверх), наверху – стеки потоков, основной стек растет вниз. Пространство между кучей и стеками обычно свободно.

Как видно, адресное пространство процесса делится на сегменты: - **Сегмент кода** (text segment) – хранит исполняемый машинный код программы (и только для чтения). - **Сегмент данных** – глобальные/статические переменные. Делится на инициализированные (initialized data) и неинициализированные (BSS) части. - **Heap (куча)** – динамически выделяемая память. Начинается сразу после BSS и может расширяться вызовами `brk()` [27]. - **Stack (стек)** – для каждого потока свой; основной стек обычно располагается вверху адресного пространства и растет вниз навстречу куче [26]. - **Библиотеки** – при загрузке динамических библиотек (.so или dll) они отображаются в адресное пространство, как правило, в промежутке между кучей и верхним адресом или в специальных зонах. - **Память mmap** – отдельные выделения через `mmap` (например, для больших буферов, или память, отображенная из файлов) могут располагаться в свободных областях адресного пространства. - **Kernel space** – важно заметить, что у каждого процесса на 64-бит системах обычно верхняя часть адресного пространства (например, адреса выше `0x7fffffffffff`) зарезервирована под ядро. Процесс в пользовательском режиме не имеет к ней доступа, но при переходе в режим ядра (системный вызов) эти адреса используются ядром (см. раздел про пользовательское/ядровое пространство ниже).

Lazy allocation (отложенное выделение памяти): Многие ОС применяют ленивое выделение памяти. Например, вызов `malloc(100 МБ)` вернет указатель, но не сразу забронирует физическую память под 100 МБ. Ядро просто отмечает в таблице страниц, что диапазон адресов виртуально выделен, но физические страницы не привязаны. Реальное выделение происходит при первом обращении к каждой странице – возникает page fault, ОС на лету выделяет физическую страницу (обычно заполненную нулями) и помечает страницу в таблице как присутствующую [28]. Это позволяет запрашивать большие блоки памяти, не тратя физические ресурсы, пока они не нужны. Также такой механизм связан с *overcommit* – ядро может зарезервировать суммарно больше виртуальной памяти, чем имеется RAM+SWAP, в расчете, что не вся она будет использована. Однако это чревато тем, что при исчерпании памяти срабатывает OOM Killer (другой аспект управления памятью).

Отложенное выделение применяется в Linux по умолчанию (режим overcommit=0 или 1). Например, сразу после `malloc(100MB)` RSS процесса не увеличится на 100MB, а вырастет постепенно по мере реального использования (чтения/записи) этих страниц [29] [30].

Разработчику важно помнить, что после `malloc` доступ к памяти лучше сразу инициализировать, чтобы фактически зарезервировать страницы (иногда используют `calloc`, так как оно не только обнуляет память, но и гарантирует загрузку страницы при записи нулей, избегая сюрпризов с отложенным выделением).

Рост кучи и стека: Куча расширяется системными вызовами (`sbrk` / `mmap`) по запросам аллокатора (например, `malloc`). Стек автоматически расширяется при использовании (до некоторого предела): если программа пытается использовать больше памяти, чем текущий размер стека, страница ниже текущего стека не помечена, происходит page fault – ядро проверяет, находится ли это сразу под существующим стеком и не превышает ли ресурс лимит (ulimit -s), и выделяет страницу под продолжение стека. Таким образом стек растет постепенно. Если же он переполнит отведенный лимит или столкнется с другими сегментами – произойдет *stack overflow* (сегфолт).

Итак, **виртуальная память** обеспечивает процессы изолированным адресным пространством, **таблицы страниц** сопоставляют виртуальные адреса с физическими, **TLB** ускоряет это сопоставление, **heap и stack** – динамические области памяти, растущие навстречу друг другу. Благодаря ленивому выделению и paging, память используется эффективно, а большие адресные пространства x64 позволяют избежать столкновения сегментов (куча и стек обычно разделены гигабайтами адресов вплоть до выработки RAM).

5. Разделение пространства пользователя и ядра

Современные операционные системы разделяют выполнение кода на два уровня привилегий: **режим пользователя (user mode)** и **режим ядра (kernel mode)**. Им соответствует разделение адресного пространства на **пользовательское пространство** (user space) и **пространство ядра** (kernel space) [31].

Зачем это нужно: Это фундаментальный механизм защиты и стабильности системы. Код, выполняющийся в режиме пользователя, имеет ограниченные привилегии – он не может напрямую обращаться к аппаратному обеспечению, к памяти других процессов, вызывать произвольные инструкции, влияющие на систему. Если пользовательская программа аварийно завершится, это не обрушит всю систему – сработает защита памяти, ОС освободит ресурсы процесса. Код же, работающий в режиме ядра, обладает полными привилегиями: он может выполнять любые инструкции CPU, управлять памятью, устройствами и пр. Разделение необходимо, чтобы **предотвратить злонамеренные или ошибочные действия пользовательского кода** – программам запрещено напрямую, минуя ядро, лезть в железо или чужую память [32] [33].

Как разделение реализовано: На уровне CPU существуют режимы или "кольца" (ring 3 – пользовательский, ring 0 – ядро на x86). Пространство адресов процесса, как упомянуто, тоже делится: часть верхних адресов зарезервирована под ядро и недоступна пользователю. Программы работают с виртуальными адресами только в своем диапазоне, бит привилегии страницы (U/S bit) в таблице страниц указывает, доступна ли страница в user mode или только в kernel mode. Любая попытка выполнить привилегированную инструкцию (например, напрямую управлять прерываниями, портами I/O) или обратиться к области ядра из user mode приводит к исключению защиты (General Protection Fault).

Взаимодействие через системные вызовы: Чтобы пользовательская программа могла воспользоваться услугами ядра (например, доступ к файлу, сеть, создание процесса), существуют

системные вызовы. Это контролируемый способ перейти в режим ядра. В Linux системный вызов осуществляется специальной инструкцией (например, `syscall` на x86_64), которая переключает процессор в режим ядра и передает управление на определенную функцию-обработчик в ядре. При этом CPU автоматически: 1. Переключается на привилегированный стек ядра (отдельный стек на каждое приложение, хранящийся в kernel space). 2. Блокирует реакцию на некоторые прерывания или меняет уровень привилегий. 3. Начинает выполнять код ядра (считая, что параметры вызова были переданы в оговоренных регистрах). После выполнения сервисной функции ядро возвращает управление обратно через инструкцию `iretq` или эквивалент, процессор возвращается в режим пользователя.

Например, когда вы вызываете в Python `open('file.txt')`, под капотом это приведет к системному вызову `openat` ядра. Интерпретатор формирует параметры и вызывает низкоуровневую функцию ОС. В результате ядро, работая с привилегиями, открывает файл и возвращает дискриптор.

Создание процесса (Fork) и потока (Clone) на уровне ядра:
- На UNIX-системах (Linux, macOS) новый процесс создается с помощью системного вызова `fork()`. Этот вызов выполняется в режиме ядра: ядро создает новую запись в таблице процессов, копирует (не полностью, а лениво, через *copy-on-write*) память родителя, дублирует дескрипторы файлов и т.д. Возвращаются два потока исполнения: родителю `fork` возвращает PID ребенка, а ребенку – 0, после чего оба продолжают выполнение. Чаще после fork сразу идет `exec()` – загрузка в дочерний процесс нового образа программы. `fork()` + `exec()` в связке примерно эквивалентно по функциональности вызову `CreateProcess` в Windows (где новый процесс сразу загружает указанный образ). - На Windows нет fork, там `CreateProcess` – это системный вызов, который создает процесс, загружает исполняемый файл, настраивает новое окружение и запускает его с нуля. - Создание нового *потока* в Linux осуществляется с помощью системного вызова `clone()` (pthread-библиотека оборачивает его). `clone` позволяет указать, какие ресурсы разделять с родителем: память, дескрипторы, таблицу файлов и т.д. Когда мы вызываем в Python `threading.Thread().start()`, внутри интерпретатора (через `_thread` модуль или C API) вызывается `pthread_create`, а тот – `clone` в ядро. Ядро создает новый поток выполнения в рамках того же процесса: новый поток получает свой стек (в пределах адресного пространства процесса) и начинает исполнение с указанной функции. В Windows аналог – `CreateThread` API, создающий поток в адресном пространстве процесса.

Привилегии кода в каждом пространстве:
- В *режиме ядра* код (ядро ОС и драйверы) может делать с системой всё, что угодно: обращаться к любому адресу памяти (хоть к памяти процесса, хоть напрямую к устройству), управлять аппаратным обеспечением, прерываниями, DMA и т.д. Поэтому код ядра должен быть высоконадежным – баг в драйвере способен повесить всю систему. - В *режиме пользователя* код изолирован: у него нет доступа к страницам ядра; он не может изменять критичные регистры CPU; все операции с оборудованием или глобальными ресурсами должны идти *только через системные вызовы*.

Таким образом достигается **разграничение ответственности и безопасности**: пользовательский код работает «под надзором» ядра. Python-программы обычно полностью в user space, за исключением переходов в ядро при обращении к файлам, сети и прочему. Разработчику Python редко нужно думать о системных вызовах напрямую – их выполняют встроенные функции (файл I/O, создание процессов и потоков и т.п.). Но под капотом каждое создание процесса через `multiprocessing` делает системный вызов `fork` (Unix) или `CreateProcess` (Windows), а каждый запуск потока – `pthread_create` (Unix) или

`CreateThread` (Win). Без перехода в режим ядра невозможно создать ни процесс, ни поток, так как только ядро может внести соответствующие изменения в системные структуры (таблицы процессов, страничные таблицы, планировщики и т.д.).

6. Разница между процессами и потоками (в CPython) + TLB

Процессы и потоки – два подхода к параллельному выполнению кода, но отличаются по архитектуре и реализации.

Архитектурные отличия: - **Процесс** – имеет **собственное** адресное пространство. Каждый процесс изолирован: память, дескрипторы файлов (если не делиться явно через `dup` / наследование) – свои. Процессы общаются между собой через IPC. В ОС процесс – полноценная сущность с собственным PID, собственным набором ресурсов. Переключение между процессами включает переключение контекста памяти (загрузка нового CR3 регистра, что обычно приводит к сбросу/смене множества записей TLB), сохранение/восстановление регистров CPU, смену пространства ядра и т.д. - **Поток** – **разделяет память с другими потоками** своего процесса. Все потоки внутри одного процесса видят одну и ту же кучу, глобальные переменные и т.п. У каждого потока свой стек и счетчик инструкций, но они проходят планирование на CPU ядром примерно так же, как процессы. В Linux потоки представлены как «облегченные процессы» (lightweight process) – по сути это записи в таблице задач с общим адресным пространством. Переключение между потоками одного процесса не требует смены адресного пространства (таблицы страниц остаются те же) – поэтому происходит быстрее, чем переключение между разными процессами.

Translation Lookaside Buffer и влияние на переключения: Когда планировщик переключает выполнение между двумя потоками *одного и того же процесса*, они пользуются одной таблицей страниц. Это значит, что кеш переводов адресов (TLB) по большей части может остаться валидным – обращения к тем же страницам не потребуют заново наполнения TLB (TLB-hit вероятны). Если же переключение идет на поток другого процесса, то меняется таблица страниц (CR3), и большинство старых TLB-строк не подходят (кроме случаев с PCID или совпадением адресов с теми же физ.страницами). Проще говоря, **переключение между потоками в рамках процесса дешевле с точки зрения памяти**, чем между процессами – у последних чаще будут промахи TLB и, возможно, выгрузка кэшей данных из-за изоляции памяти.

GIL – влияние на многопоточность в CPython: В стандартной реализации Python (CPython) существует **Глобальная блокировка интерпретатора (Global Interpreter Lock)**, которая не позволяет выполнять байт-код Python более чем одному потоку одновременно внутри одного процесса. Это сделано из-за особенностей управления памятью в CPython (отсутствие потокобезопасного сборщика мусора без блокировок). Следствие: **многопоточность в Python не увеличивает вычислительную производительность на CPU-задачах** – все треды поочередно выполняются на одном ядре (переключаясь между собой, обычно каждые 100 операций байт-кода). Таким образом, если у вас задача чисто расчетная (CPU-bound) и вы запустите 10 потоков, они поделят между собой один CPU и не ускорят вычисление (а иногда и замедлят из-за оверхеда переключений).

Многопроцессность (несколько процессов Python) не имеет этого ограничения – у каждого процесса свой интерпретатор и GIL. Поэтому 10 процессов могут выполняться на 10 ядрах параллельно [1]. Именно поэтому модуль `multiprocessing` позволяет «разгрузить» GIL и воспользоваться многоядерностью.

Однако: многопоточность не бесполезна в Python – для задач ввода-вывода (сетевых, дисковых) GIL освобождается во время системных вызовов (например, чтение сокета освобождает GIL, пока ждем данные). Кроме того, многие библиотечные операции на C уровне сами снимают GIL (например, сжатие, обработка изображений, numpy вычисления) – тогда потоки могут реально параллельно работать. Поэтому для I/O-bound проблем (веб-запросы, ожидание файлов) Python-потоки отлично подходят, позволяя перекрывать ожидание одного потока вычислениями в другом.

Ограничения потоков: Разделяемая память – палка о двух концах. С одной стороны, потоки легко поделиться данными (глобальные объекты видны всем), не нужны очереди/IPC. С другой – это приводит к типичным проблемам многопоточности: гонки данных, необходимость синхронизации (Lock/Mutex) при совместном доступе к изменяемым объектам, сложности отладки. Один неаккуратный поток может повредить общую структуру и испортить данные другим. В процессах же по умолчанию данные независимы – если один упал или повредил свою память, другие не страдают (если только не использовали общую разделенную память).

Ограничения процессов: Накладные расходы. Создать процесс – дороже, чем поток: нужно дублировать таблицу страниц, выделить отдельную память, запустить новый интерпретатор. Межпроцессной обмен – через сериализацию и копирование, тогда как потоки могут просто вызывать общие функции и передавать указатели на общие объекты. 10 процессов Python будут потреблять существенно больше памяти (10 интерпретаторов, каждая со своими копиями модулей, объектов) по сравнению с 10 потоками в одном процессе. Контекстный переключатель между процессами тоже тяжелее, как обсуждали (сброс TLB и т.д.). Также, создание большого числа процессов упирается в ограничения ОС (количество дескрипторов, pid и прочее) быстрее, чем потоков. Однако слишком много активных потоков тоже неэффективны (из-за рассредоточенности времени CPU).

TLB hits/misses и производительность: Для длительно работающих потоков в общем адресном пространстве, TLB примерно одинаково заселен нужными записями. При переключении между ними (например, ОС чередует потоки процесса) TLB в целом «теплый». Если же в системе постоянно мелькают десятки разных процессов, каждый со своим набором страниц, TLB успевает часто промахиваться – CPU тратит больше времени на таблицы страниц. В реальности ОС старается по возможности выполнять на ядре некоторое время один и тот же процесс (*time slice*), чтобы кэш и TLB нагрелись под него.

Как узнать объем TLB: Как уже упоминалось, OS не предоставляет простого способа, но можно посмотреть документацию CPU. Например, для Intel x86_64: L1 TLB может иметь 64 записи для данных и 64 для инструкций (4KB страниц), плюс отдельный для больших страниц; L2 TLB – 1024 записей и т.д. Эти числа меняются от модели. Знание объема TLB нужно разве что в системном программировании или производительном коде (например, при оптимизации прохода по большим массивам важно учитывать, сколько страниц укладывается в TLB).

В обычной практике Python-разработчика эти нюансы не видны напрямую, но они объясняют, почему, скажем, 10 процессов могут работать чуть медленнее, чем идеальный линейный масштаб – из-за накладных на кэш/TLB, и почему контекст-переключения дорого обходятся.

Особенности реализации потоков в CPython: Каждый `threading.Thread` управляется планировщиком ОС, Python не делает «green threads» (в отличие от async, который не создает системных потоков). Поэтому Python-потоки – настоящие системные потоки. GIL же серийно запускает их выполнение на CPU. Есть альтернативные реализации Python без GIL (Jython,

IronPython, или экспериментальное ветвление CPython с отключаемым GIL), но в стандартном интерпретаторе на 2023-2025 годы GIL все еще существует, хотя идут работы над его устранением (PEP 703).

Сводная таблица – процессы vs потоки:

- **Изоляция:** процессы изолированы (память не общая, защищена); потоки разделяют память процесса. Поэтому процессы надёжнее (падение одного не сломает другие), а потоки требуют тщательной синхронизации при общем доступе.
- **Создание и ресурс:** создание процесса тяжелее (копирование памяти, запуск интерпретатора), поток легче (просто новая точка выполнения в существующем процессе). Процесс занимает больше памяти (отдельная куча, стеки, объекты), поток – только свой стек плюс незначительные структуры.
- **Многопроцессорность:** процессы (в Python) могут параллельно выполняться на разных ядрах (нет общего GIL), потоки – **в Python именно для CPU-bound** – не масштабируются на несколько ядер из-за GIL [1] . (В других языках или в случае I/O-bound это не так строго, но мы говорим про CPython.)
- **Коммуникация:** у процессов – через IPC (очереди, пайпы, сокеты, общая память и пр.), у потоков – через общие переменные (но нужны блокировки). Обмен данными между процессами обычно медленнее (serialization), между потоками – быстрее (просто передача ссылки/указателя, но риск гонок).
- **Использование в Python:** Для вычислительно сложных задач – `multiprocessing` (или `ProcessPoolExecutor`), для задач ожидания (скачивание множества веб-страниц, работа с файлами) – `threading` или `ThreadPoolExecutor`. Например, веб-сервер на Python часто обслуживает запросы потоками, потому что каждый запрос может ждать I/O, и GIL не мешает параллельно обрабатывать другие запросы. А вот рендеринг изображения или перебор огромной комбинации лучше распараллелить на процессы.

Отметим, что есть альтернатива – **asyncio** (асинхронность без реальных потоков), но это выходит за рамки темы.

7. Сравнение многопоточности и многопроцессности в Python (плюсы и минусы)

Наконец, сопоставим преимущества и недостатки подходов с учетом особенностей CPython:

Плюсы многопоточности: - Минимальные накладные расходы на создание и переключение. Потоки легче и создаются быстрее, чем процессы. - Используют общую память – легко делиться данными между задачами (не нужно сериализовать/копировать объекты). Например, общий кэш в памяти доступен всем потокам напрямую. - Подходят для большого числа мелких задач: сотни потоков могут существовать (с учётом, что в Python активными одновременно будет ограниченное число из-за GIL) – но по памяти они гораздо дешевле сотен процессов. - Отлично масштабируют задачи ввода-вывода: пока один поток ждёт сеть или диск, другой выполняется. GIL отпускется при I/O, поэтому десятки потоков могут параллельно ждать/обрабатывать результаты, эффективно используя одно ядро (или несколько ядер, поскольку во время I/O GIL свободен для других) – т.е. throughput по запросам растет.

Минусы многопоточности: - **GIL (в CPython)** – главный минус для вычислительных задач. Только один поток исполняет Python байт-код в единицу времени [1] . Поэтому на чисто CPU-нагрузке 8 потоков не быстрее 1 (хотя есть выход – использовать расширения на C,

освобождающие GIL, но это отдельный разговор). - Сложность разработки: нужно защищать общие данные. Классические проблемы: deadlock (взаимная блокировка), race conditions (состояния гонки). Отладка многопоточных багов крайне трудна. - Если один поток вызвал ошибку, по умолчанию (если не перехвачена) она может завершить весь процесс – т.е. упадут все потоки сразу, т.к. процесс един. - Потоки разделяют адресное пространство, значит любой из них может теоретически испортить общую кучу (например, вызвать сегфолт через внешний Си-модуль) и это повлияет на всех. - Масштабирование по CPU ограничено – не выйдешь за предел 100% одного ядра (в CPython).

Плюсы многопроцессности: - **Полноценный параллелизм на всех ядрах.** Каждый процесс – независимый интерпретатор Python, свой GIL. Можно задействовать 4 ядра на 100% с 4 процессами (или более на более задач). - Высокая изоляция: процесс ошибся – упал только он (главный процесс может отследить и перезапустить при необходимости). Память не разделяется, нет риска повредить данные соседнего процесса багом. - Простейший способ обойти GIL и ускорить вычисления – использовать `multiprocessing` или `ProcessPoolExecutor` для нагрузок, легко масштабируется по ядрам. - Можно даже распределять на разные машины (если использовать сети) – хотя это уже более сложные кластеры. - В CPython `multiprocessing` позволяет относительно прозрачно передавать объекты и получать результаты (автоматически сериализуя) – удобство в том, что код написаный для синхронного выполнения, можно относительно легко адаптировать под процессы.

Минусы многопроцессности: - **Накладные расходы по памяти и времени:** Запуск процесса – более тяжелая операция (некоторые оценки: ~ десятки миллисекунд против миллисекунд у потока). Каждый процесс потребляет память за счет своего интерпретатора и копий объектов. Например, если у вас 8 процессов, каждый импортирует одни и те же модули, то они загружены 8 раз (кроме случаев использования `fork` на UNIX, где поначалу память копируется лениво и общие объекты могут разделяться до модификации – но на Windows даже это недоступно, там spawn и все загружается с нуля). - **Межпроцессный обмен медленнее:** Нужно сериализовать объекты (pickle) для отправки через Pipe/Queue, что копирует данные. Большие объемы данных тяжело гонять между процессами. В поточной программе можно просто передать ссылку на список или разделить глобальную структуру – в процессах так не выйдет (есть механизмы разделяемой памяти, но они требуют больше управления). - **Ограниченная численность:** Сотни тяжелых процессов запустить трудно – упрутся в лимиты и ресурсы ОС. Хотя процессы лучше изолированы, но планировщик ОС не рассчитан на слишком большое число активных процессов (так же, впрочем, как и на огромное число потоков). - Усложняется отладка и дизайн: например, в `multiprocessing` функции должны быть сериализуемы, код защищать блоком `if __name__=="__main__":` и т.п. Не все лямбды или локальные функции можно просто так передать процессу. Ошибки в дочерних процессах не видны напрямую (они возникают в отдельном процессе, нужно обрабатывать исключения через `multiprocessing` механизмы).

Когда что лучше применять:
- Если задача сильно загружает CPU и распараллеливается – используйте **процессы** (например, обработка изображений, научные вычисления без GIL-friendly библиотек). `multiprocessing.Pool` или `ProcessPoolExecutor` упростят распараллеливание. - Если задача в основном ждет ввода-вывода или сетевых ответов – можно использовать **потоки**. Например, веб-скрейпер, делающий тысячи HTTP-запросов: GIL тут не мешает, а 100 потоков упростят код (по сравнению с 100 процессами, которые съедят больше памяти). - Если нужен высокий параллелизм и при этом много совместных данных, и GIL не проблема (например, задачи манипуляции с большим общим массивом numpy, который **освобождает GIL**) – тогда **потоки** удобнее, т.к. могут работать с общим numpy-массивом без копирования. - Если

необходимо надежное изолирование (например, выполнять произвольные недоверенные вычисления) – **процессы** предпочтительнее, чтобы сбойный код не повлиял на главный сервис.

В конкретных приложениях зачастую комбинируют подходы: например, веб-сервер может использовать несколько процессов (воркеры) для масштабирования на ядра, а каждый воркер внутри – пулы потоков для обслуживания запросов.

Пример компромисса – GIL: В Python нейронные сети (библиотека GIL-free, например, TensorFlow, PyTorch) могут параллельно загружать данные потоками (I/O) и параллельно обучаться на нескольких GPU (процессы) – таким образом используют и потоки, и процессы.

Подытожим, **многопроцессность vs многопоточность**: - Многопроцессность позволяет использовать несколько CPU полноценно (обходя GIL), обеспечивает лучшую устойчивость к сбоям, но требует более сложного обмена данными и потребляет больше ресурсов [34]. - Многопоточность проста в запуске и быстрый обмен данными, экономна по памяти, но ограничена GIL (в CPython) и сложна из-за синхронизации, не дает изоляции.

Оба подхода – инструменты в арсенале разработчика. Senior Python developer должен уметь выбрать правильный: оценить характер нагрузки (CPU или I/O, объем данных для обмена, требуемая изоляция, доступность GIL для данной задачи) и на основе этого применять либо `threading` / `concurrent.futures.ThreadPoolExecutor`, либо `multiprocessing` / `ProcessPoolExecutor`. Часто на практике начинают с более простой реализации (например, потоков для сетевой задачи) и только при выявлении узкого места (например, GIL на CPU) переключаются на процессы.

Примечание: В Python 3.11+ идут работы по улучшению многопоточности (возможно, GIL будет опциональным в будущих версиях), но на 2025 год при написании этого документа GIL по-прежнему актуален, и все вышеприведенные соображения сохраняют силу.

---

[1] [10] [11] [14] [15] [16] multiprocessing — Process-based parallelism — Python 3.13.3 documentation
https://docs.python.org/3/library/multiprocessing.html

[2] [3] What exactly is Python multiprocessing Module's .join() Method Doing? - Stack Overflow
https://stackoverflow.com/questions/25391025/what-exactly-is-python-multiprocessing-modules-join-method-doing

[4] [5] Fork vs Spawn in Python Multiprocessing - British Geological Survey
https://britishgeologicalsurvey.github.io/science/python-forking-vs-spawn/

[6] [9] concurrent.futures — Launching parallel tasks — Python 3.13.3 documentation
https://docs.python.org/3/library/concurrent.futures.html

[7] [8] python - How does ThreadPoolExecutor().map differ from ThreadPoolExecutor().submit? - Stack Overflow
https://stackoverflow.com/questions/20838162/how-does-threadpoolexecutor-map-differ-from-threadpoolexecutor-submit

[12] [13] [17] [18] python - Multiprocessing - Pipe vs Queue - Stack Overflow
https://stackoverflow.com/questions/8463008/multiprocessing-pipe-vs-queue

[19] [21] [22] [23] Linux Memory Management: Understanding Page Tables, Swapping, and Memory Allocation | Linux Journal
https://www.linuxjournal.com/content/linux-memory-management-understanding-page-tables-swapping-and-memory-allocation

[20] pmap(1)
https://docs.oracle.com/cd/E19253-01/816-5165/6mbb0m9o2/index.html

[24] Translation Lookaside Buffer (TLB) in Paging | GeeksforGeeks
https://www.geeksforgeeks.org/translation-lookaside-buffer-tlb-in-paging/

[25] Loongson-3A5000-H-4.19.0-17-loongson-3.cpuinfo - GitHub
https://github.com/ThomasKaiser/sbc-bench/blob/master/results/cpuinfo/Loongson-3A5000-H-4.19.0-17-loongson-3.cpuinfo

[26] Memory layout for a typical process
https://www.qnx.com/developers/docs/8.0/com.qnx.doc.neutrino.sys_arch/topic/dll_Process_memory.html

[27] Memory Layout of C Programs | GeeksforGeeks
https://www.geeksforgeeks.org/memory-layout-of-c-program/

[28] [30] Linux Virtual Memory - overcommit
https://www.polarsparc.com/xhtml/VM-OverCommit.html

[29] linux - Are some allocators lazy? - Stack Overflow
https://stackoverflow.com/questions/864416/are-some-allocators-lazy

[31] [32] [33] User space and kernel space - Wikipedia
https://en.wikipedia.org/wiki/User_space_and_kernel_space

[34] How to use a Python multiprocessing module | Red Hat Developer
https://developers.redhat.com/articles/2023/07/27/how-use-python-multiprocessing-module