ChatGPT

# Многопроцессное и многопоточное программирование в Python (с деталями Linux)

В этом документе подробно рассмотрены механизмы работы с процессами и потоками в Python на уровне ядра Linux. Будут приведены примеры кода, объяснены ключевые методы и продемонстрировано взаимодействие с системой (мониторинг памяти, просмотр карт памяти и т.д.). Материал рассчитан на Senior-уровень: глубокие технические детали, примеры и ссылки на источники.

## 1. Работа с `multiprocessing.Process`

Класс `Process` из модуля `multiprocessing` позволяет создать новый процесс (аналог `fork()` в ОС). Основные методы и атрибуты:

- `start()` – запускает процесс (фактически выполняет `fork()` или `spawn`). После вызова создаётся дочерний процесс, который выполняет целевую функцию.
- `join(timeout=None)` – ожидает завершения процесса (блокирует до конца работы или указанного таймаута). Позволяет синхронизировать процессы.
- `is_alive()` – возвращает `True`, если процесс всё ещё выполняется (после `start()` и до завершения).
- `terminate()` – принудительно останавливает процесс (отправляет ему SIGTERM). Код завершения будет отрицательным (по сигналу) [1].
- `exitcode` – код завершения процесса (`None`, если ещё не завершился). После `join()` можно проверить, как завершился процесс (0 — нормальное, отрицательный — прерван сигналом) [2].

**Пример:** создаём два процесса, проверяем их состояние и выходные коды.

```
from multiprocessing import Process
import time, signal

def worker():
    print("Worker started, PID", os.getpid())
    time.sleep(1)
    print("Worker finished")

if __name__ == "__main__":
    p = Process(target=worker)
    print("Запуск процесса")
    p.start()                               # запускаем процесс
    print("Процесс запущен, is_alive():", p.is_alive())
    p.join()                                # ждём завершения
    print("Процесс завершён, exitcode =", p.exitcode)  # 0 если всё хорошо
```

```
    # Демонстрация terminate()
    p2 = Process(target=worker)
    p2.start()
    p2.terminate()                          # принудительно прерываем
    p2.join()
    print("Second process exitcode:", p2.exitcode)  # будет отрицательным
(например, -15)
```

*При использовании* `Process` *важно: методы* `start()`, `join()`, `is_alive()`, `terminate()` *и обращение к* `exitcode` *должен делать родительский процесс (создавший экземпляр)* [1] [2].

Примечание: для корректной работы на Windows и при использовании `spawn`-метода нужно оборачивать создание процессов в `if __name__ == "__main__":` (чтобы главный модуль без рекурсии импортов мог создать процессы) [3].

## 2. Модуль `concurrent.futures`

Модуль `concurrent.futures` предоставляет высокоуровневый интерфейс для параллельного запуска задач. Для процессов используется `ProcessPoolExecutor`, для потоков – `ThreadPoolExecutor`. Основные методы:

- `submit(fn, *args)` – запускает функцию `fn` асинхронно, возвращает объект `Future`.
- `map(fn, iterable)` – похож на встроенный `map`: применяет `fn` к каждому элементу входного списка **параллельно**, возвращая итератор по результатам в *порядке исходной последовательности* [4].
- `as_completed(futures)` – функция-модуль, которая принимает набор `Future`-объектов и возвращает их в порядке завершения задач [4].

**Разница** `map` **vs** `as_completed`**:** При использовании `executor.map()` результаты возвращаются в **том же порядке**, что и входные данные [4]. Даже если одна задача завершится раньше, результат всё равно ждёт своего места. В отличие от этого, `concurrent.futures.as_completed(futures)` выдаёт завершившиеся задачи сразу по мере готовности (без сохранения исходного порядка) [4]. Это удобно для реактивной обработки результатов по мере их готовности.

```
from concurrent.futures import ProcessPoolExecutor, as_completed

def square(x):
    time.sleep(0.1)
    return x * x

nums = [3, 1, 4, 2]
with ProcessPoolExecutor() as executor:
    # map: результаты в исходном порядке
    results = list(executor.map(square, nums))
    print("executor.map:", results)
    # submit + as_completed: по мере готовности
    futures = [executor.submit(square, x) for x in nums]
```

```
    for fut in as_completed(futures):
        print("as_completed result:", fut.result())
```

Этот пример показал бы, что `executor.map` возвращает `[9, 1, 16, 4]`, несмотря на разное время сна, а `as_completed` выдаёт, например, `1, 4, 9, 16` (в порядке фактического завершения) [4].

Основные выводы о `concurrent.futures`:
- `ProcessPoolExecutor` позволяет легко распараллеливать CPU-bound задачи на разные процессы.
- `map` удобен для простого применения функции к списку, но он блокирует порядок результатов.
- `as_completed` даёт больше гибкости: можно реагировать на завершение задач мгновенно.

## 3. Механизмы взаимодействия между процессами

Для обмена данными между процессами в Python используют различные каналы.

- **Очереди** (`multiprocessing.Queue`): потокобезопасная очередь FIFO. Любой объект, помещённый в `Queue.put()`, **сериализуется** (обычно через `pickle`) и пересылается. Метод `get()` возвращает воссозданный объект (новую копию) [5]. Пример:

```
from multiprocessing import Process, Queue

def producer(q):
    data = {"val": 123}
    q.put(data)

def consumer(q):
    item = q.get()
    print("Получено из очереди:", item)

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 объект `data` отправляется через очередь; в дочернем процессе `consumer` он восстанавливается из байтов [5].

- **Каналы (Pipe)** (`multiprocessing.Pipe`): создаёт пару объектов `Connection`, представляющих два конца канала. По умолчанию канал двунаправленный. Оба конца имеют методы `send(obj)` и `recv()`. Как и у `Queue`, при `send()` объект сериализуется, а при `recv()` восстанавливается [6]. Пример:

```
from multiprocessing import Process, Pipe
```

```python
def send_msg(conn):
    conn.send("Привет через Pipe")
    conn.close()

if __name__ == "__main__":
    parent_conn, child_conn = Pipe()
    p = Process(target=send_msg, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # Получает строку
    p.join()
```

Два конца `Pipe()` – как две трубки: по одной можно читать, по другой писать. Так, родитель отправляет по `send()`, а ребёнок читает через `recv()` [6]. Важно: если два процесса пытаются одновременно читать или писать с одного конца, данные могут повредиться; поэтому один конец обычно только для чтения, другой – только для записи [7].

- **Сокеты** (`socket`): стандартные TCP/IP или UNIX-сокеты можно использовать для IPC (особенно, если процессы могут быть на разных хостах или исполняются через локальную сеть). Простейший пример – запуск сервера в одном процессе и клиента в другом через `socket.socket(AF_UNIX)` или `AF_INET`. Хотя этот способ более низкоуровневый, он гибок.

В итоге, для безопасного IPC в Python обычно рекомендуют использовать `Queue` или `Pipe` [8]. Эти структуры уже обеспечивают сериализацию и межпроцессную безопасность.

## 4. Выделение памяти процессу в Linux

Каждому процессу в Linux выделяется виртуальное адресное пространство, состоящее из сегментов (код, данные, куча, стек, mmap). Физическая память при этом резервируется **лениво** (demand paging) [9] [10]. То есть ОС не выделяет физические страницы заранее, а только при реальном обращении к ним (приём «ленивого выделения»).

- **Виртуальная и физическая память:** Каждому процессу назначается виртуальное пространство (например, 4 ГБ на 32-бит системе) [10]. ЦПУ вместе с MMU переводит виртуальные адреса в физические через таблицу страниц. Таблица страниц – это структура, хранящая соответствия (виртуальная→физическая) [10] [11]. При обращении к адресу процессор проверяет TLB – кэш недавних переводов. Если адрес в TLB (TLB hit), используется сохранённая запись; при промахе (TLB miss) происходит обращение к таблице страниц и последующее обновление TLB [12]. Это ускоряет работу памяти.

- **Просмотр используемой памяти:** Команда `ps aux` показывает два важных столбца: **VSZ** (виртуальный размер) и **RSS** (размер в физической памяти, «Resident Set Size»). VSZ – объём всего выделенного адресного пространства (включая неиспользуемые страницы) [9]. Он может быть большим, но не говорит о фактическом использовании RAM (часто страницы не загружены в память, а только зарезервированы). RSS – число реально загруженных в RAM страниц (сумма приватной + долевой). Однако RSS переоценивает используемую память из-за разделяемых библиотек (общие страницы считаются для каждого процесса) [13]. Иными словами, VSZ показывает максимальный потенциал использования памяти, а RSS – текущую загрузку (с учётом дублирования общих сегментов) [9] [13].

- **Инструменты мониторинга:** `top` и `htop` также выводят VSZ/RSS. Команда `pmap PID` детально показывает карту памяти процесса: её сегменты, размер в КБ, права доступа и тип (например, [anon] – анонимный, heap, stack, библиотеки) [14] [15]. Вывод `pmap -x PID` даст таблицу с виртуальными адресами, VSZ, RSS и состоянием (dirty, права, имя мэппинга) [16]. Например, `[anon]` в выводе `pmap` – это кусок памяти, не взятый из файла (обычно куча или стек), выделяемый «на лету» [15]. Кроме того, можно напрямую смотреть `/proc/[PID]/maps` (или `smaps`): там перечислены все сегменты процесса с их правами и физическими страницами. Утилита `vmstat` выдаёт общесистемную статистику памяти/переключений страниц, что полезно при анализе нагрузки, но для одного процесса хватит `ps`, `pmap` и `/proc`.

- **Рост кучи и стека:** Обычно при создании процесса (`fork`) ему не выделяется сразу весь стек/куча, они растут по требованию. Увеличение кучи можно заметить по увеличению RSS после большого `malloc` или заполнения списков. Аналогично, рост стека виден через `/proc/[PID]/maps` (адреса стекового сегмента увеличиваются) или косвенно через `pmap`.

- **Copy-on-write (COW):** При `fork()` память копируется лениво: фактические копии страниц происходят только при записи в них. Поэтому изначально родитель и ребёнок разделяют одни и те же физические страницы до тех пор, пока один из них не попытается их изменить [10]. Это позволяет быстро создавать новые процессы без мгновенного расхода памяти.

## 5. User Space и Kernel Space

В Linux (и других ОС) применяется **разделение адресного пространства** на ядро (kernel space) и пространство пользователя (user space) [17]. Это фундаментальный механизм безопасности и стабильности:

- **Почему важно разделение:** Процессы в user space (пространстве пользователя) не имеют прямого доступа к памяти ядра или других процессов. Если пользовательский код случайно/намеренно повреждает память, ядро останется изолированным и сможет отреагировать на ошибку. Разделение обеспечивает **аппаратную защиту**: доступ к привилегированному участку разрешён только ядру. Это предотвращает многие типы багов и атак [17].

- **Практический пример:** Когда приложение читает файл или обращается к устройству, оно делает системный вызов, переключаясь в режим ядра для выполнения привилегированных операций. Весь остальной код (бизнес-логика, библиотеки) работает в user space, с ограниченными правами. Такой подход гарантирует, что даже если приложение падает или вызовет ошибку доступа памяти, ядро и другие процессы не пострадают.

## 6. Создание процесса и потока на уровне ОС

В Linux/Unix новые процессы и потоки создаются разными системными вызовами:

- **fork()** – стандартный POSIX-вызов: создаёт точную копию (клона) текущего процесса. После `fork()` и в родителе, и в ребёнке исполняется тот же код, но возвращаемое значение

`fork()` указывает, где мы: 0 в дочернем, PID дочернего в родительском. `fork()` производит ленивое копирование памяти (copy-on-write) [10]. Обычно после `fork()` сразу следует `exec()` (замена образа процесса на новый), чтобы запустить другую программу.

- **exec()** – набор семейства вызовов (например, `execve()`): загружает новый исполняемый файл в текущий процесс, заменяя старое содержимое. Таким образом можно после `fork()` запустить другую программу.

- **clone()** – специфичный для Linux вызов (ядра). Позволяет создавать новый процесс или поток с гибкой настройкой разделяемых ресурсов (можно делить файловые дескрипторы, память и т.д.). На самом деле `fork()` и `pthread_create()` под капотом обычно вызывают `clone()` с разными флагами.

- **pthread_create()** – POSIX-вызов создания **потока** в рамках существующего процесса. Поток (thread) начинает выполняться с заданной функции, разделяя с другими потоками одного процесса единое адресное пространство.

**Как Python создает процессы/потоки:**
Python-модуль `threading` использует под капотом pthreads (в CPython стандартной реализации) [18]. То есть `threading.Thread` создает нативный поток ОС (Pthread), и он «живёт» внутри текущего процесса, разделяя память. При этом из-за GIL одновременно байткод Python выполняет только один поток (см. далее).

Модуль `multiprocessing` на Linux по умолчанию использует метод *fork*: он вызывает `fork()`, поэтому в дочернем процессе унаследуется копия памяти (copy-on-write) и запустится код целевой функции [19]. На Windows и macOS по умолчанию используется `spawn`: Python запускает новый интерпретатор и импортирует модуль целевой функции [19]. Фактически при `spawn` процесс создаётся через вызов `fork()` + `exec()`, чтобы получить свежий интерпретатор.

## 7. Разница между процессами и потоками

Основные различия **по памяти и изоляции**:

- **Память:** Каждый процесс имеет своё виртуальное адресное пространство (отдельную кучу, свой стек) и независимый набор дескрипторов. Между процессами изолирована память (общие данные нужно передавать через IPC) [20]. В отличие от этого, потоки одного процесса разделяют общее адресное пространство: доступ к глобальным переменным, куче и т.д. у разных потоков общий. Это облегчает обмен данными между потоками (достаточно обращаться к общей переменной), но требует синхронизации (мьютексы, блокировки) для предотвращения гонок.

- **GIL:** В CPython есть **GIL** (Global Interpreter Lock) – глобальная блокировка интерпретатора. Она гарантирует, что в каждый момент времени активен лишь один поток выполняет байткод Python [18]. Из-за этого многопоточность CPython не даёт ускорения для вычислений, нагружающих CPU – только конкарентность. Многопроцессность же обходит GIL: каждый процесс имеет свой GIL, поэтому код может реально параллельно выполняться на разных ядрах [18] [21].

- Threads: один GIL → нет истинного параллелизма для CPU-bound задач [18].

- Processes: каждый процесс со своим интерпретатором → параллелизм на многоядерных системах.

- **Контекстные переключения:** Потоки «легче» переключать, потому что общий адресный контекст остаётся тем же; переключение между потоками стоит меньше, чем между процессами. Процессы же имеют раздельную память, и для переключения нужно сменить более объёмный контекст (страница памяти, TLB-флеш, etc.).

- **TLB и процессы:** При переключении процессов часто сбрасывается TLB (кеш адресных преобразований), что делает такие переключения дороже. Нити же разделяют одинаковую таблицу страниц, поэтому у них выше шанс TLB hit, чем при переходе между разными процессами.

- **Использование ресурсов:** Поток создаётся быстрее и дешевле по памяти (разделяем одну память) [20] . Процесс затратнее: создаётся новое пространство, данные нужно копировать (или не копировать из-за COW, но всё равно отдельный RSS, отдельные структуры).

## 8. Плюсы и минусы `multithreading` vs `multiprocessing`

- **Многопоточное программирование (threads):**
- *Плюсы:* Низкая стоимость создания и переключения. Удобная передача данных через общую память. Подходит для I/O-bound задач (сети, диск), где CPU простаивает (GIL не мешает, так как потоки часто ждут I/O) [22] [23] .

- *Минусы:* Защита разделяемых данных – необходимость блокировок, возможные гонки. GIL препятствует ускорению CPU-bound: при тяжёлых вычислениях производительность не растёт [18] [21] .

- **Многопроцессное программирование (processes):**

- *Плюсы:* Истинный параллелизм на многоядерных системах, поскольку процессы выполняются независимо [21] . Отсутствие общей памяти делает задачи безопаснее (изоляция исключений). Каждый процесс свой GIL, можно масштабироваться по ядрам.
- *Минусы:* Высокие накладные расходы на создание процессов и передачу данных между ними (надо сериализовать объекты для `Queue` / `Pipe`). Больше потребление памяти (каждый процесс имеет свой RSS). Контекст-переключения дороже, чем у потоков.

**Когда что использовать:**
- Для **CPU-bound** задач (численные вычисления, научные расчёты) обычно эффективнее *multiprocessing* (или `concurrent.futures.ProcessPoolExecutor`) – каждый процесс нагрузит отдельное ядро [21] .
- Для **I/O-bound** задач (работа с сетью, файловыми системами, базами данных) подойдёт *multithreading* – потоки будут блокироваться на ожидании и не держат GIL, что позволяет работать эффективно без создания лишних процессов [22] [23] .
- `multiprocessing` полезен, когда требуется полностью изолированное выполнение (надежность, устранение влияния GIL). `multithreading` удобен, когда нужны быстрые «легкие» задачи и быстрый обмен данными, но не требуется CPU-параллелизм.

Таким образом, выбор зависит от характера задачи: ограничения GIL и природа нагрузки (CPU vs I/O) обычно определяют, что лучше – нити или процессы [21] [20] .

**Источники:** документация Python и статьи по теме многопоточности/многопроцессности в сочетании с Linux-механизмами [5] [18] [9] [15] [21].

---


[1] [2] [3] [5] [6] [7] [8] multiprocessing — Process-based parallelism — Python 3.13.3 documentation
https://docs.python.org/3/library/multiprocessing.html

[4] multithreading - Python's `concurrent.futures`: Iterate on futures according to order of completion - Stack Overflow
https://stackoverflow.com/questions/16276423/pythons-concurrent-futures-iterate-on-futures-according-to-order-of-completi

[9] [13] ps output - Difference between VSZ vs RSS memory usage - LinuxConfig
https://linuxconfig.org/ps-output-difference-between-vsz-vs-rss-memory-usage

[10] [11] landley.net
https://landley.net/writing/memory-faq.txt

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

[14] [15] [16] How to analyze a Linux process' memory map with pmap
https://www.redhat.com/en/blog/pmap-command

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

[18] [23] threading — Thread-based parallelism — Python 3.13.3 documentation
https://docs.python.org/3/library/threading.html

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

[20] Beyond Threads & Processes: Unlocking the Power of Python’s Shared Memory | by Elshad Karimov | Medium
https://elshad-karimov.medium.com/beyond-threads-processes-unlocking-the-power-of-pythons-shared-memory-5eb9eb79d110

[21] [22] Python Multithreading vs. Multiprocessing Explained | Built In
https://builtin.com/data-science/multithreading-multiprocessing