

Multitasking

Alexander Evgin

10 апреля 2020 г.

Outline

1 Parallelism vs concurrency

2 Processes and threads

- Threads synchronization

- Processes synchronization

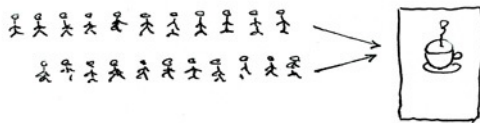
- GIL

3 Asynchronous functions

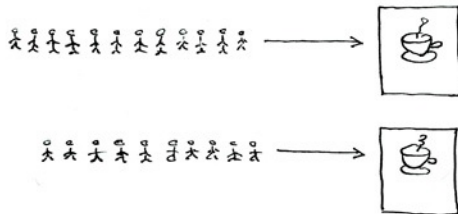
Task execution

Parallelism vs Concurrency

Concurrent = Two Queues One Coffee Machine



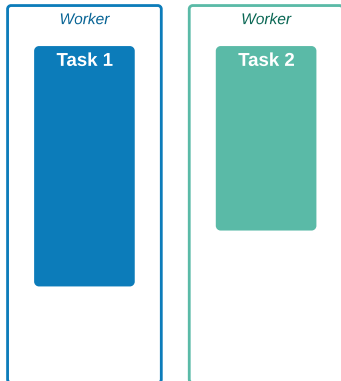
Parallel = Two Queues Two Coffee Machines



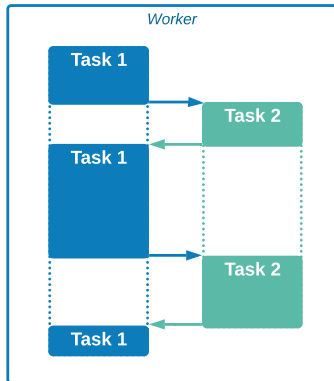
© Joe Armstrong 2013

Parallelism vs Concurrency

Parallelism



Concurrency



Process

- **Процесс** — запущенная программа.
- Окружение процесса:
 - виртуальное адресное пространство
 - указатель на исполняемую инструкцию
 - стек вызовов
 - таблица дескрипторов
- У каждого процесса своё **изолированное** окружение.
- Процессы удобны для одновременного выполнения нескольких задач.
- Процессы могут создавать другие (дочерние) процессы.
- Альтернативный способ: делегировать каждую задачу на выполнение **поток**у.

Process

Linux:

- Создание процесса: системный вызов `fork`
- Дочерние процессы наследуют окружение от родительских
- Иерархия процессов

Windows NT:

- Создание процесса: системный вызов `NtCreatProcess`
- Дочерние процессы имеют произвольное требуемое окружение

Thread (поток, тред, “нить”)

- **Поток** — единица выполнения задачи. Чаще всего поток находится “внутри” процесса. Его исполнение происходит независимо от других потоков и процессов.
- Поток разделяет все ресурсы процесса. У нескольких потоков одного процесса общие данные и системные ресурсы.
- Потоки удобны для одновременного выполнения нескольких задач, которым требуется доступ к разделяемому состоянию (общей памяти).

Processes and threads

- Потоки и процессы — единицы объекты операционной системы. Операционная система управляет их выполнением (Process Scheduling).
- Реализация зависит от операционной системы.
- Эффективность зависит от ресурсов системы. Единица выполнения — ядро процессора.

Модуль subprocess

Позволяет создавать подпроцессы (дочерние процессы).
Каждый из них — самостоятельная программа со своим окружением.

main.py

```
1 import subprocess as sp
2
3 proc = sp.run(['py', 'child.py'])
```

child.py

```
1 print('Hello, world!')
```

Модуль multiprocessing

Интерфейс позволяет запускать функции как подпроцессы.

```
1 import multiprocessing as mp
2
3 def greeting(name: str):
4     print(f'Hello, {name}')
5
6 if __name__ == "__main__":
7     child = mp.Process(target=greeting, args=('User',))
8     child.start()
```

- subprocess — удобнее вызывать сторонние (не python) приложения
- multiprocessing — удобнее производить параллельные вычисления в одной программе

Модуль threading

Позволяет создавать новые потоки. Поток в Python — это системный поток, его выполнением управляет не интерпретатор, а операционная система.

```
1  from threading import Thread
2
3  def greeting(name: str):
4      print(f'Hello, {name}')
5
6  t = Thread(target=greeting, args=('User',))
7  t.start()
```

Имя и идентификатор потока

При создании потоку можно указать имя. По умолчанию оно "Thread-N":

```
>>> Thread().name
'Thread-1'
>>> Thread(name="NumberCruncher").name
'NumberCruncher'
```

У каждого активного потока есть идентификатор — неотрицательное число, уникальное для всех активных потоков.

```
>>> t = Thread()
>>> t.start()
>>> t.ident
4350545920
```

Присоединение потоков

Один поток может дожидаться исполнения другого. Выполнение потока приостановится, пока не завершится поток `t`.

Метод `join()`:

```
>>> t = Thread(target=time.sleep, args=(5, ))
>>> t.start()
>>> t.join() # блокируется на 5 секунд
>>> t.join() # выполняется моментально
```

Норма: корневой поток дожидается всех остальных. В таких случаях происходит корректное завершение всех потоков (заккрытие файлов, освобождение семафоров). Python *не имеет* встроенного корректного завершения потоков.

Присоединение потоков

Проверить, выполняется ли поток, можно с помощью метода `is_alive()`:

```
>>> t = Thread(target=time.sleep, args=(5, ))
>>> t.start()
>>> t.is_alive()
True
>>> t.is_alive() # через 5 секунд
False
```

Синхронизация потоков

У потоков в одном процесса общая память. Следовательно, несколько потоков могут менять одно и тоже значение. Это может привести к непредсказуемому поведению (*undefined behavior*).

Примитивы синхронизации:

- Мьютекс — выставляемый флаг (0 или 1). Используется для обеспечения эксклюзивного доступа к разделяемому состоянию.
- Семафор — численный счетчик. Позволяет захватывать и освобождать себя нескольким потокам сразу (увеличивать и уменьшать свое значение на 1).
- ...

Синхронизация потоков: мьютекс

Мьютекс `threading.Lock`:

```
1 class SharedCounter:
2     def __init__(self, value):
3         self.value = value
4         self._lock = Lock()
5
6     def increment(self, delta=1):
7         self._lock.acquire()
8         self.value += delta
9         self._lock.release()
```

`acquire()` блокируется до тех пор, пока счетчик не станет свободным (другой поток не вызовет `release()`).

Синхронизация потоков: событие

С помощью `threading.Event` можно организовать ожидание некоторого “события” одним или более потоками:

```
1  io_ready = Event()
2
3  def initialize_logging():
4      io_ready.wait()
5      # ...
6
7  def initialize_disk_io():
8      # ...
9      io_ready.set()
```

Все задачи с `initialize_logging` будут ждать, пока не выполнится `initialize_disk_io`.

Синхронизация потоков: условные переменные

`threading.Condition` используется для отправки сигналов между потоками. Метод `wait()` блокирует вызывающий поток, пока какой-то другой поток не вызовет метод `notify()` или `notify_all()`.

```
1  q = deque()
2  is_empty = Condition()
3
4  def producer():
5      while True:
6          is_empty.acquire()
7          q.append(...)
8          is_empty.notify()
9          is_empty.release()
10
11 def consumer():
12     while True:
13         is_empty.acquire()
14         while not q:
15             is_empty.wait()
16         ... = q.popleft()
17         is_empty.release()
```

Синхронизация потоков: пример

Функция `follow` читает сообщения из переданного ей соединения и кладёт их в очередь на обработку.

```
1  def follow(connection, connection_lock, q):
2      try:
3          while True:
4              connection_lock.acquire()
5              message = connection.read_message()
6              connection_lock.release()
7              q.put(message)
8      except InvalidMessage:
9          follow(connection, connection_lock, q)
10
11  follower = Thread(target=follow, args=...)
12  follower.start()
```

Синхронизация потоков: пример

Функция `follow` читает сообщения из переданного ей соединения и кладёт их в очередь на обработку.

```
1  def follow(connection, connection_lock, q):
2      try:
3          while True:
4              connection_lock.acquire()
5              message = connection.read_message()
6              connection_lock.release()
7              q.put(message)
8      except InvalidMessage:
9          follow(connection, connection_lock, q)
10
11 follower = Thread(target=follow, args=...)
12 follower.start()
```

Вопрос: что может пойти не так?

Синхронизация потоков

Чтобы методы `acquire()` и `release()` не засоряли код и не порождали ошибки, все примитивы синхронизации поддерживают протокол контекстного менеджера:

```
1 def follow(connection, connection_lock, q):
2     try:
3         while True:
4             with connection_lock:
5                 message = connection.read_message()
6                 q.put(message)
7     except IOError:
8         follow(connection, connection_lock, q)
```

Модуль queue: safe-thread

Модуль queue реализует несколько потокобезопасных очередей:

- Queue — FIFO очередь,
 - LifoQueue — LIFO очередь aka стек,
 - PriorityQueue — очередь, элементы которой — пары вида (priority, item).
-
- Никаких особых изысков в реализации очередей нет: все методы, изменяющие состояние, работают “внутри” мьютекса.
 - Класс Queue использует в качестве контейнера deque, а классы LifoQueue и PriorityQueue — список.

Модуль queue: пример

```
def worker(q):  
    while True:  
        item = q.get() # блокирующе ожидает следующий  
        do_something(item)  
        q.task_done() # уведомляет очередь о выполнении  
                       # задания  
  
def master(q):  
    for item in source():  
        q.put(item)  
  
q.join() # блокирующе ожидает, пока все элементы  
         # очереди не будут обработаны
```


Синхронизация процессов

- Разные окружения, нет общей памяти.
- Операционная система предоставляет механизм взаимодействия: ***Inter-process communication (IPC)***:
 - Signal
 - Named pipe / Anonymous pipe
 - Shared memory
 - Socket / Unix domain socket
 - Message queue (*probably the best*)
 - ...

Синхронизация процессов

```
1  proc = subprocess.run(['py', 'child.py'],  
2                          capture_output=True)  
3  print(proc.stdout)  # b'Hello, world!'
```

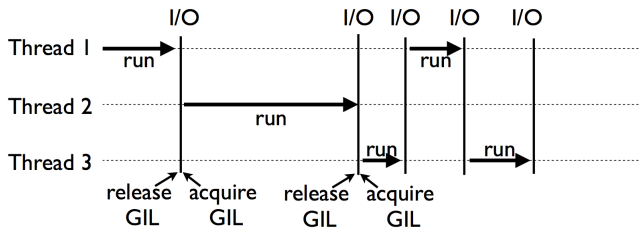
```
1  from multiprocessing import Process, Value  
2  
3  def func(val):  
4      val.value += 1  
5  
6  v = Value('i', 0)  
7  Process(target=func, args=(v,)).start()
```

GIL

Global Interpreter Lock (GIL)

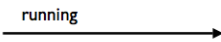
GIL — глобальный мьютекс интерпретатора, который гарантирует, что в каждый момент времени только один поток имеет доступ к внутреннему состоянию интерпретатора.

- GIL — это фича интерпретатора CPython.
- **Идея:** В CPython всегда выполняется только один поток (на самом деле нет).
- **Но:** GIL может быть освобожден (released) с помощью Python C API: например, все операции ввода/вывода в CPython отпускают GIL.

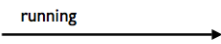


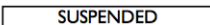
GIL mechanism

- Предположим, что программа использует только один поток. GIL принадлежит потоку. Ничего интересного не происходит.

Thread 1 

- Что произойдёт при появлении второго потока? Ничего: GIL всё ещё принадлежит первому потоку. Второй поток должен каким-то образом его получить.

Thread 1 

Thread 2 

GIL mechanism

- Второй поток ожидает GIL в течении промежутка времени TIMEOUT в надежде, что первый поток сам освободит GIL, например, в результате операции ввода/вывода.

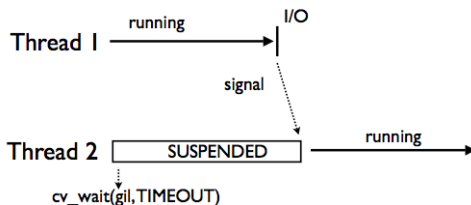
Thread 1 $\xrightarrow{\text{running}}$

Thread 2 SUSPENDED
⋮
`cv_wait(gil, TIMEOUT)`

- Далее возможны два случая в зависимости от того, отпустил первый поток GIL или нет.

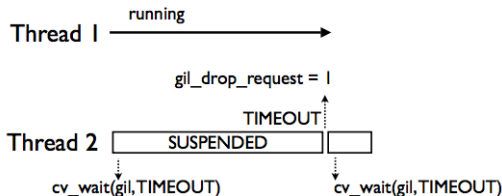
GIL mechanism

- Простой случай: первый поток отпускает GIL и сигнализирует об этом второму потоку.
- Второй поток захватывает GIL и начинает исполнение.

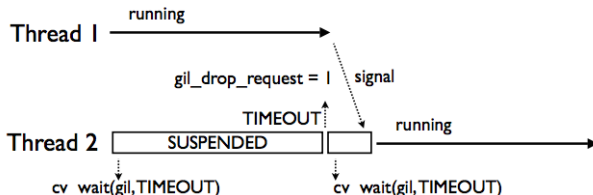


GIL mechanism

- Сложный случай: второй поток сигнализирует первому потоку о своём желании захватить GIL, устанавливает глобальный флаг `gil_drop_request` и повторяет ожидание.

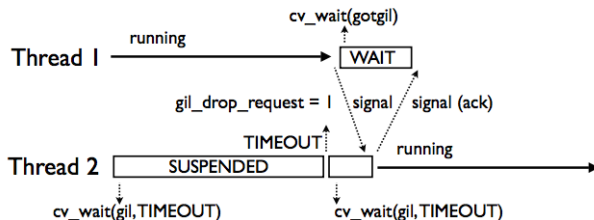


- Первый поток завершает выполнение текущей инструкции, отпускает GIL и сигнализирует об этом второму потоку.



GIL mechanism

- Первый поток ожидает уведомление об успешном захвате GIL первым потоком. Это сделано для того, чтобы предыдущий владелец GIL не мог захватить его повторно ("GIL Battle").
- Второй поток захватывает GIL, отправляет первому потоку уведомление и начинает исполнение.



Global Interpreter Lock (GIL)

Ещё раз: GIL запрещает *физическую параллельность* потоков. Это защищает от неправильной работы интерпретатора, но не защищает от вашего собственного кода:

```
counter += 1
```

(*read counter* → *calculate counter + 1* → *write self.counter*)

Плохой сценарий (в котором в каждый момент выполняется один поток):

- thread1 reads counter (0)
- thread2 reads counter (0)
- thread1 calculates number + 1 (1)
- thread2 calculates number + 1 (1)
- thread1 writes 1 to number
- thread2 writes 1 to number

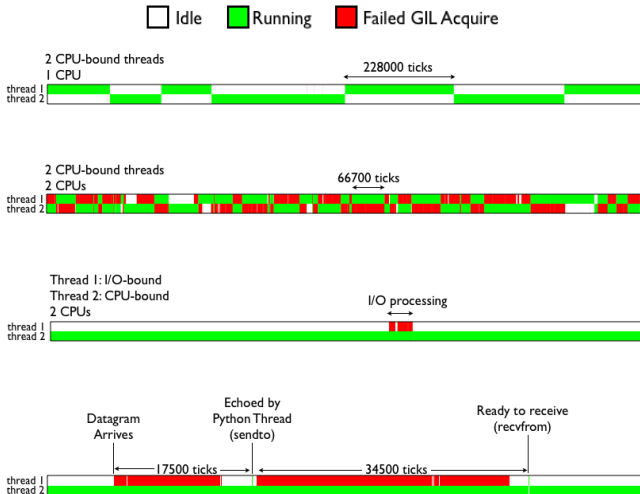
Результат: counter = 1

GIL — good or bad?

Ответ зависит от вашей задачи.

- + Однопоточный код выполняется быстрее (один Lock, JIT)
- + Решает проблему счетчика ссылок (reference counter)
- + GIL не мешает использовать потоки для конкурентности при работе с вводом/выводом
- + Упрощение интеграции с C, который не thread-safe
- Невозможность использовать потоки в Python для параллелизма: несколько потоков не ускоряют, а иногда даже замедляют работу программы
- “GIL Battle” — несколько потоков бьются за владение GIL

GIL Battle



David Beazley

Why hasn't the GIL been removed yet?

*I'd welcome a set of patches into Py3k only if the performance for a single-threaded program (and for a multi-threaded but I/O-bound program) **does not decrease**.*

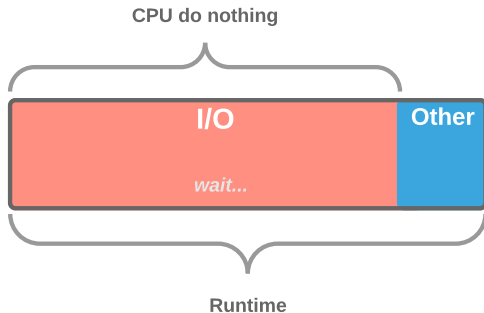
Guido van Rossum

Asynchrony

Asynchrony

Практически все программы тратят большую часть времени своей работы на I/O операции.

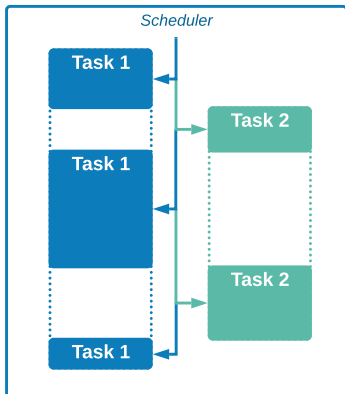
(Ядро Linux: I/O - $\approx 86\%$ CPU time, но это не точно)



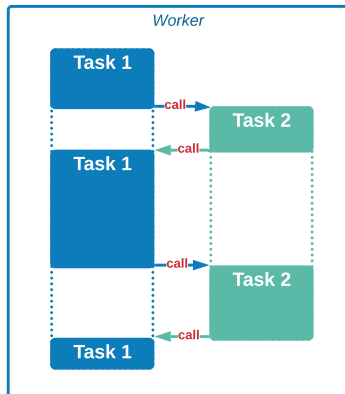
Asynchrony

Асинхронное выполнение в Python — это кооперативная многозадачность.

Preemptive multitasking



Cooperative multitasking



Асинхронные паттерны

У исполняемой задачи может быть два состояния: вычисление и ожидание. Ожидание — самое время, чтобы передать выполнение другой задаче.

- Корневая задача — цикл исполнения
- Отслеживание “готовности к выполнению” в задаче (ready to read/write, системный вызов)
- Переключение контекста выполнения между задачами (selector-ы / генераторы)

Асинхронность в Python

- Асинхронность на системных вызовах готовности (`select`, `event loop`)
- Асинхронность на генераторах и сопрограммах (`coroutine`, `yield from`)
- Модуль `asyncio`, синтаксис `async/await`

Интерфейс asyncio

```
1  import asyncio
2
3  async def foo():
4      print('foo...1')
5      await asyncio.sleep(0.1)
6      print('foo...2')
7      await asyncio.sleep(0.3)
8      print('foo...3')
9
10 async def bar():
11     print(' bar...1')
12     await asyncio.sleep(0.2)
13     print(' bar...2')
14     await asyncio.sleep(0.1)
15     print(' bar...3')
16
17 ioloop = asyncio.get_event_loop()
18 ioloop.run_until_complete(asyncio.wait([foo(), bar()]))
```