# Лекция №9

## Потоки и процессы

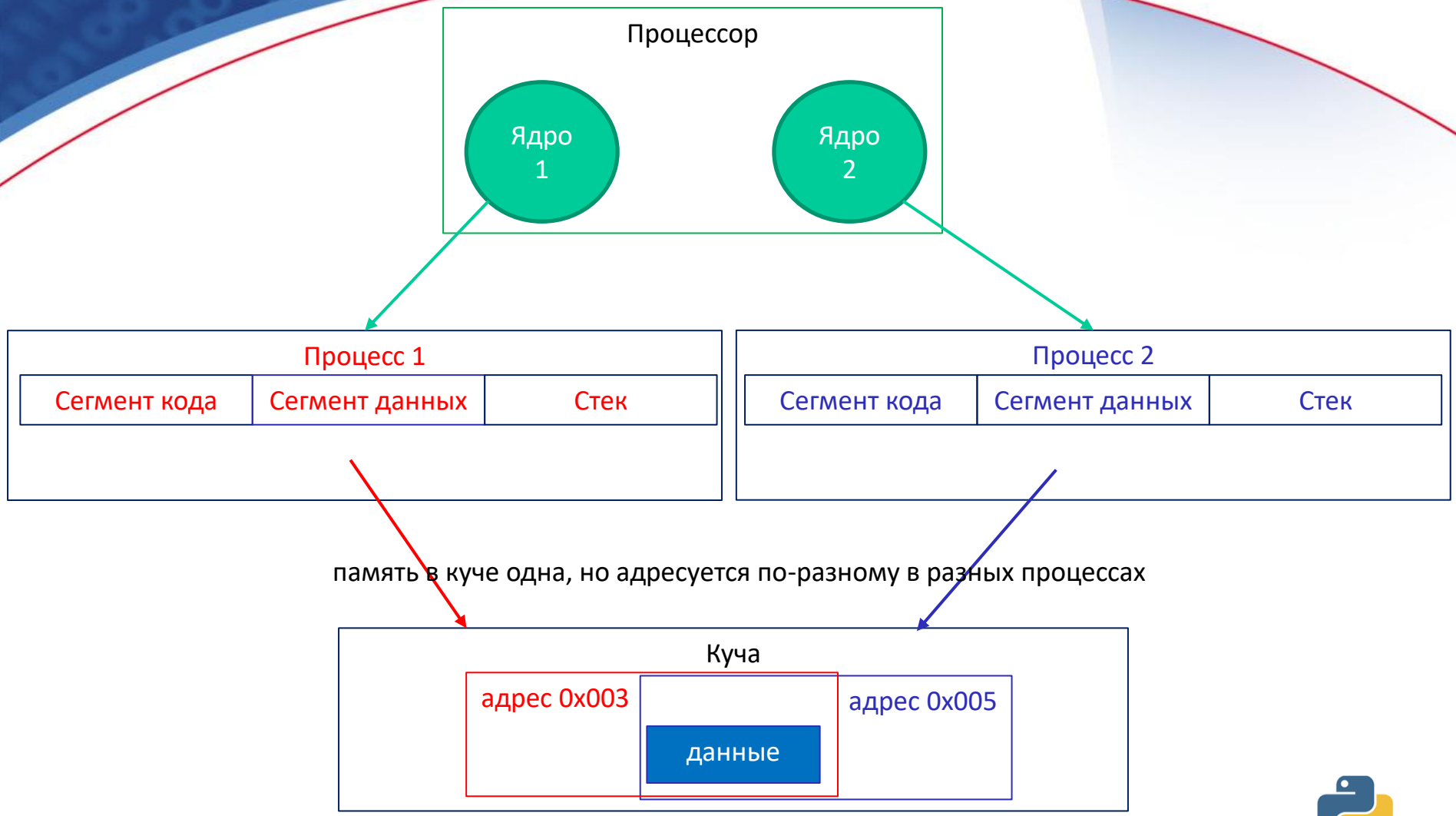
- Общая информация
- Процессы
- Потоки
- Многопоточность
- Модуль threading
- Межпоточное взаимодействие
- Объекты синхронизации
- Потокобезопасная очередь
- GIL – Global Interpreter Lock
- Green threads
- Asyncio
- Многопроцессность
- Межпроцессное взаимодействие
- Синхронизация процессов
- Использование разделяемой памяти
- Создание пула процессов
- Практика

# Общая информация

С появлением многоядерных процессоров стала общеупотребительной практика распространять нагрузку на все доступные ядра. Существует два основных подхода в распределении нагрузки: использование процессов и потоков. Процессы и потоки связаны друг с другом, но при этом имеют существенные различия. Вы представляете, что такое поток и процесс?

# Процесс

Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.

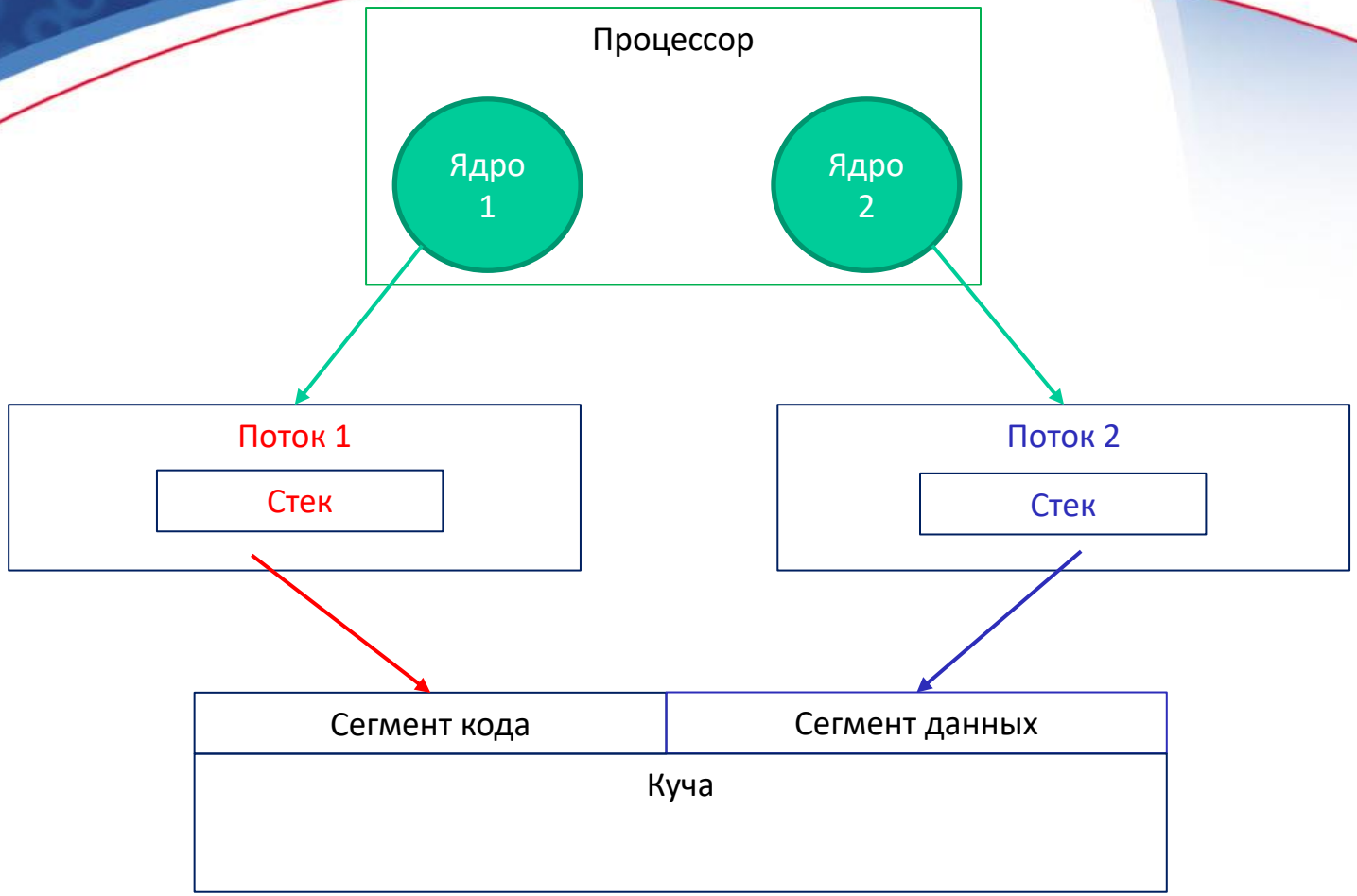

Процессор
Ядро 1
Ядро 2
Процесс 1
Сегмент кода
Сегмент данных
Стек
Процесс 2
Сегмент кода
Сегмент данных
Стек
память в куче одна, но адресуется по-разному в разных процессах
Куча
адрес 0x003
данные
адрес 0x005

НИИТ

# Поток

Поток — это объект выполнения внутри процесса, включающий в себя набор последовательных операций, состояние и ресурсы. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса.

Поток использует то же самое пространства стека, что и процесс, а множество потоков совместно используют данные своих состояний. Как правило, каждый поток может работать (читать и писать) с одной и той же областью памяти, в отличие от процесса, который не может просто так получить доступ к памяти другого процесса. У каждого потока есть собственные регистры и собственный стек, но и другие потоки могут их использовать. Отсюда возникают проблемы управления доступом к разделяемым ресурсам в многопоточных процессах, необходимость синхронизации и предотвращения взаимных блокировок.


Процессор
Ядро 1
Ядро 2
Поток 1
Стек
Поток 2
Стек
Сегмент кода
Сегмент данных
Куча

# Многопоточность

Что дает распараллеливание программы на несколько потоков?
Рассмотрим следующую задачу:

```
import random
import time

def compute(number):
    # что-то долго вычисляем
    print('Старт вычислений №{}'.format(number))
    sleeping = random.randint(1, 5)
    time.sleep(sleeping)
    print('Конец вычислений №{}. Затрачено {} секунд'.format(number, sleeping))

start = time.perf_counter()
# считаем что-то много раз с разными параметрами
for i in range(5):
    compute(i)
print('Общее время вычислений в секундах: {}'.format(int(time.perf_counter() - start)))
```

# Многопоточность

Все операции в программе (т.е. все вызовы функции compute) выполняются последовательно, общее время выполнения фактически складывается из времени выполнения функции compute при каждом вызове.

```
Старт вычислений №0
Конец вычислений №0. Затрачено 3 секунд
Старт вычислений №1
Конец вычислений №1. Затрачено 4 секунд
Старт вычислений №2
Конец вычислений №2. Затрачено 1 секунд
Старт вычислений №3
Конец вычислений №3. Затрачено 4 секунд
Старт вычислений №4
Конец вычислений №4. Затрачено 4 секунд
Общее время вычислений в секундах: 16
```

# Многопоточность

Попробуем применить многопоточность, используя модуль threading, обеспечив параллельное выполнение каждого вызова функции compute.

```
import threading
import random
import time

def compute(number):
    # что-то долго вычисляем тем же способом
    print('Старт вычислений №{}'.format(number))
    sleeping = random.randint(1, 5)
    time.sleep(sleeping)
    print('Конец вычислений №{}. Затрачено {} секунд'.format(number, sleeping))

start = time.perf_counter() # считаем что-то много раз с разными параметрами
threads = []
for i in range(5):
    thr = threading.Thread(target=compute, args=(i,))
    thr.start()
    threads.append(thr)

for thr in threads:
    thr.join()
print('Общее время вычислений в секундах: {}'.format(int(time.perf_counter() - start)))
```

# Многопоточность

В результате мы получаем, что несмотря на то, что суммарное время выполнения функции compute при каждом вызове такое же, как и в примере без многопоточности (16 секунд), время выполнения всей программы равно всего 5 секундам. Т.е. не сумме времен всех вызовов функции compute, а только максимальному времени выполнения этой функции. Таким образом, с применением многопоточности наша программа работает более чем в 3 раза быстрее.

```
Старт вычислений №0
Старт вычислений №1
Старт вычислений №2
Старт вычислений №3
Старт вычислений №4
Конец вычислений №4. Затрачено 1 секунд
Конец вычислений №1. Затрачено 3 секунд
Конец вычислений №3. Затрачено 3 секунд
Конец вычислений №0. Затрачено 4 секунд
Конец вычислений №2. Затрачено 5 секунд
Общее время вычислений в секундах: 5
```

# Многопоточность

При работе с потоками обязательные следующие операции:

- создание потока
- старт потока
- ожидание завершения потока

Изначально у каждой программы есть один поток - он же "главный". Этот поток создается операционной системой при запуске процесса. С точки зрения программиста он почти не отличается от созданных вручную. Практически же существуют некоторые особенности, например, именно этот поток реагирует на системные прерывания (например, нажатие Ctrl+C), и пока выполняется "неглавный" поток, программа не будет на них реагировать.

# threading

Как уже было сказано ранее, для организации многопоточности в Python используется модуль threading. Существует два способа создания потоков с использованием этого модуля: передачей исполняемой функции в конструктор и наследованием.

```
# Первый вариант:

def f(a, b, c):
    # do something
    pass

th = threading.Thread(name='th', target=f, args=(1, 2), kwargs={'c': 3})
# name - имя потока. Ни на что не влияет, но может быть полезно при отладке.
# target - точка входа (любой callable object, функция, связанный метод класса).
# args - позиционные аргументы.
# kwargs - именованные аргументы.
```

Поток с именем 'th1' будет создан, но не запущен. После запуска будет вызвана функция f с параметрами a=1, b=2, c=3. Все аргументы могут быть опущены.

# threading

```
# Второй вариант:

class MyThread(threading.Thread):
    def __init__(self, a, b, c):
        threading.Thread.__init__(self)
        self.a = a
        self.b = b
        self.c = c

    def run(self):
        # do something, using self.a, self.b, self.c
        pass


# Результат практически тот же самый, но в новом потоке будет запущен метод run.
th = MyThread(1, 2, 3)
```

# threading

После того, как поток создан, его нужно запустить. В обоих случаях это делается через вызов:

```
th.start()
```

Любой поток рано или поздно нужно завершить. Делается это простым выходом из функции потока. Не существует ПРАВИЛЬНОГО способа завершить поток снаружи. Это - принципиальное ограничение. Т.е. если вы хотите завершить поток из другого - просигнализируйте ему о своей просьбе (выставив флаг-переменную, например). Поэтому после сигнала о завершении нужно дождаться, когда поток реально закончит свою работу. Делается это так:

```
th.join()
```

Метод join приостановит выполнение потока, вызвавшего его, и будет ждать когда поток th завершит свое выполнение. Зачастую поток, стартовавший th, его же и ждет, но бывают и исключения.

# threading

Еще раз про работу с потоками:

- Потоки можно создавать и запускать.
- Можно просить их закончить свою работу, но нельзя останавливать принудительно.
- Завершения потока нужно дожидаться.

# Межпоточное взаимодействие

Основная проблема многопоточного программирования – разделение ресурсов. Потоки должны взаимодействовать между собой, или, другими словами, изменять состояние разделяемых между ними объектов. Несколько потоков могут пытаться изменить один объект одновременно, и результат будет непредсказуем. Объекты могут быть сложными… К примеру, запись в объект "пользователь" фамилии одним потоком, а имени другим может привести к неожиданному результату (например, фамилия перетрет имя, или наоборот). Существуют и более разрушительные примеры.

Предположим, что есть два потока, имеющих доступ к общему списку. Первый поток может делать итерацию по этому списку:

```
for x in my_list:
```

а второй в этот момент начнет удалять значения из этого списка. Тут может произойти все что угодно: программа может упасть, или мы просто получим неверные данные.

# Межпоточное взаимодействие

С этой точки зрения все объекты (переменные) разделяются на:

- Неизменяемые. Если объект никто не меняет, то синхронизация доступа ему не нужна. К сожалению, таких не очень много.
- Локальные. Если объект не виден остальным потокам, то доступ к нему синхронизировать тоже не требуется.
- Разделяемые и изменяемые. Синхронизация необходима.

Синхронизация доступа к объектам осуществляется с помощью объектов синхронизации. Рассмотрим основные из них.

# Объекты синхронизации: блокировки (мьютексы)

Простейший объект синхронизации – блокировка (мьютекс):

```
import threading

class Point:
    def __init__(self):
        self._mutex = threading.RLock()
        self._x = 0
        self._y = 0

    def get(self):
        with self._mutex:
            return (self._x, self._y)

    def set(self, x, y):
        with self._mutex:
            self._x = x
            self._x = y
```

# Объекты синхронизации: блокировки (мьютексы)

Работает все это так: - при вызове метода захватываем мьютекс через with self._mutex: - весь код внутри with блока будет выполнятся только в одном потоке. Другими словами, если два разных потока вызовут .get то пока первый поток не выйдет из блока второй будет его ждать - и только потом продолжит выполнение.

Зачем это все нужно? Координаты нужно менять одновременно - ведь точка это цельный объект. Если позволить одному потоку поменять x, а другой в это же время изменит y, логика алгоритма может оказаться нарушенной.

# Объекты синхронизации: блокировки (мьютексы)

Показанный в примере класс RLock (reentrant lock) - вариант простого мьютекса, с которым поток блокируется только в том случае, если мьютекс захвачен другим потоком, в то время как с обычным мьютексом (класс Lock) может заблокироваться и сам поток, захвативший этот мьютекс, если он попытается захватить тот же самый мьютекс повторно.

```
lock = threading.Lock()
lock.acquire()
lock.acquire() # вызов заблокирует выполнение
lock.release() # мы никогда не дойдем до этой строчки

lock = threading.RLock()
lock.acquire()
lock.acquire() # вызов не заблокирует выполнение
# последующий код будет выполнятся
```

# Объекты синхронизации: семафоры

Семафоры - более сложный механизм блокировок. Здесь используется уже не флаг с двумя состояниями, а счетчик. Когда число потоков, захвативших семафор, достигает заданного значения, семафор блокирует выполнение всех последующих потоков, пытающихся захватить этот семафор.

```
import threading
semaphore = threading.BoundedSemaphore(5)
semaphore.acquire() # уменьшает счетчик
# доступ к общему ресурсу
semaphore.release() # увеличивает счетчик
```

Обычно семафоры используются чтобы лимитировать доступ к ограниченному ресурсу, не требующему исключительного владения, например, к сетевым подключениям или серверу баз данных. Инициализируем семафор максимальным числом потоков, которые должны иметь доступ к ресурсу, и внутренняя реализация семафора позаботится обо всем остальном.

# Объекты синхронизации: события

Событие (Event) - простейший объект синхронизации, обеспечивающий работу с флагом состояния. Поток может ожидать установки этого флага, или устанавливать и сбрасывать его самостоятельно, а также проверять, не установлен ли флаг, перед тем как начать ожидать его установления.

```
event = threading.Event()  # изначально флаг установлен (is_set() == True)
event.set() # установка флага (is_set() == True) - в серверном потоке
event.clear() # сброс флага (is_set() == False) - в серверном потоке
event.wait() # ожидание флага (пока is_set() == False) - в клиентском потоке
event.is_set() # проверяем, установлен ли флаг, прежде чем ожидать
```

Если флаг установлен, метод wait ничего не сделает. Если флаг сброшен, wait заблокирует выполнение потока до тех пор, пока флаг вновь не будет установлен (другим потоком). Любое количество потоков могут ожидать одно и то же событие одновременно.

# Объекты синхронизации: условные переменные

Есть условные переменные (Condition) - более совершенный вариант события, фактически являющиеся обертками над мьютексами. Условная переменная позволяет реализовать ожидание уже не специализированного объекта синхронизации, а истинности обычного логического выражения. Также она позволяет управлять количеством потоков, которые можно разблокировать, при освобождении мьютекса, и устанавливать таймер ожидания. По умолчанию, использует RLock (который сама создает), но можно задать свой Lock или Rlock, передав его в конструктор.

```
cv = threading.Condition()
# захватываем item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# освобождаем item
with cv:
    make_an_item_available()
    cv.notify()
```

# Потокобезопасная очередь

Для решения означенной ранее задачи, когда один поток читает некоторый список, а другой записывает в этот список информацию, в условиях неопределенности относительно применения GIL имеет смысл воспользоваться потокобезопасной очередью, обеспечивающей синхронизацию доступа к своим данным.

Модуль queue реализует несколько потокобезопасных очередей:

- Queue — FIFO очередь,
- LifoQueue — LIFO очередь (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 двунаправленную очередь (deque), а классы LifoQueue и PriorityQueue — список.

# Потокобезопасная очередь

```
import threading
import queue

q = queue.Queue()

def worker():  # функция выполняемая в дочернем потоке
    with open('outfile.txt', 'w') as fout:
        item = '0'  # чтоб можно было зайти в цикл while
        while item.isdigit():  # non-digit строку считаем признаком окончания расчетов
            item = q.get()  # ожидаем появление элемента в очереди
            fout.write('start')
            if item.isdigit():
                fout.write(f' {int(item) ** 2} ')  # выводим в файл квадрат числа
            fout.write('finish\n')
            q.task_done()  # уведомляем очередь о завершении обработки


t = threading.Thread(target=worker)
t.start()
with open('infile.txt') as fin:
    for line in fin:
        items = line.split()  # будем анализировать строку посимвольно
        for each in items:
            if each.isdigit():
                q.put(each)  # добавляем элемент в очередь и ничего не ждем
                q.join()  # ожидаем обработки элементов в очереди
    q.put('stop')  # добавляем 'stop' в очередь и ничего не ждем,
    q.join()  # ожидаем обработки элементов в очереди
t.join()  # ожидаем завершения дочернего потока
```

# Не все так просто

Рассмотрим еще один пример работы с потоками. Сначала для однопоточного приложения:

```
import time

def count(n):
    while n > 0:
        n -= 1

c = 80000000
start = time.perf_counter()
count(c)
count(c)
print('{0:.2f} sec.'.format(time.perf_counter() - start))
```

```
13.05 sec.
```

# Не все так просто

Теперь попробуем распараллелить задачу на два потока:

```
from threading import Thread
import time

def count(n):
    while n > 0:
        n -= 1

c = 80000000
start = time.perf_counter()
t1 = Thread(target=count, args=(c,))
t2 = Thread(target=count, args=(c,))
t1.start()
t2.start()
t1.join()
t2.join()
print('{0:.2f} sec.'.format(time.perf_counter() - start))
```

```
13.77 sec.
```

T.e. в данном случае двумя потоками приложение выполняется медленнее, чем одним!

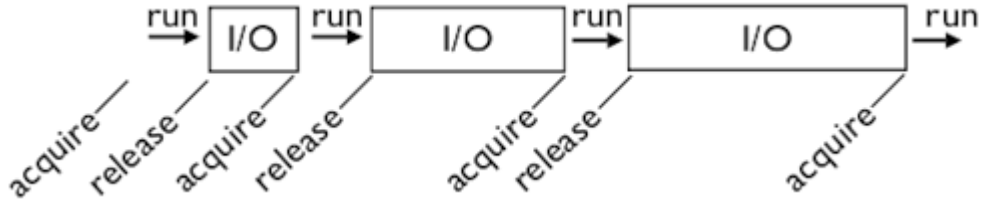
# GIL – Global Interpreter Lock

Потоки, как правило, используются в двух целях: обеспечения параллельности выполнения задач (parallelism) и независимости их выполнения друг от друга – конкурентности (concurrency). Последнее не требует обеспечения возможности одновременного выполнения этих задач (разными ядрами процессора), но предполагает возможность переключения процессора от заблокированной (ожиданием ввода пользователя, данных от сервера и т.д.) задачи к следующей, стоящей на очереди.

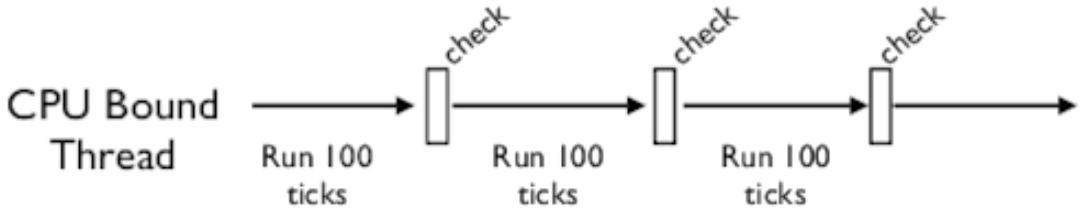
В Python есть GIL —мьютекс, который гарантирует, что в каждый момент времени только один поток имеет доступ к внутреннему состоянию интерпретатора. **Наличие GIL делает невозможным использование потоков в Python для распараллеливания расчетов (parallelism) в большинстве случаев**: несколько потоков не ускоряют, а иногда даже замедляют работу программы. Но GIL не мешает использовать потоки для конкурентности (concurrency) при работе с вводом/выводом, например, при сетевых операциях, что позволяет рассматривать потоки в Python, как способ реализации асинхронности.

# GIL – Global Interpreter Lock

Когда поток, захвативший GIL, переходит к ожиданию завершения операции ввода/вывода интерпретатор передает GIL другому потоку.




Также GIL может быть передан другому потоку, если поток, владеющий GIL, не выполняет никаких реальных операций (например, в нем запущена функция sleep) на основании periodic check, которую интерпретатор проводит каждые 100 «тиков» по внутреннему счетчику интерпретатора (это значение можно поменять с помощью sys.setcheckinterval()) для CPU-зависимых потоков, невыполняющих операции ввода/вывода.



# Green threads

Создание потоков (а тем более процессов) весьма дорогостоящая для ОС операция, как по времени, так и по памяти. К тому же в Python из-за GIL фактически сведена на нет возможность параллельного выполнения потоков на многоядерных процессорах. Итак, параллелизм при помощи потоков недостижим, а для обеспечения только конкурентности они слишком дороги. Так мы приходим к идее неких микропотоков, которые исполняются фактически в одном процессе и потоке (имеется в виду поток, созданный средствами ОС – native поток; речь о параллелизме не идет вообще), но создаются без обращения к ОС (и это обходится весьма дешево) и работать с ними так же удобно, как и с обычными native потоками. Эти микропотоки, эмулирующие native потоки, получили название “зеленые потоки” – green threads. В Python они называются гринлетами (greenlets) и реализованы на C. Для работы с гринлетами в Python есть библиотека gevent, которая до Python 3 была частью стандартной библиотеки, теперь же поддерживается, как отдельное расширение, уступив место другим способам реализации асинхронности.

# Green threads

Гринлеты хорошо зарекомендовали себя в различных системах, поэтому они все еще активно используются, позволяя создавать асинхронный код, оставаясь в парадигме многопоточного программирования.

```
import gevent
import time

def count(n):
    while n > 0:
        n -= 1

c = 80000000
start = time.perf_counter()
g1 = gevent.spawn(count, c) # создаем и запускаем 1-й гринлет
g2 = gevent.spawn(count, c) # создаем и запускаем 2-й гринлет
gevent.joinall([g1, g2]) # ожидаем завершения гринлета
print('{0:.2f} sec.'.format(time.perf_counter() - start))
```

```
13.36 sec.
```

Время выполнения меньше, чем при использовании native потоков, но больше, чем вообще без распараллеливания.

# Asyncio (определения)

Следующим этапом поддержки асинхронности в Python стала библиотека asyncio, реализующая упрощенный и универсальный (схожий с другими языками – C++, JS) подход к организации конкурентного программирования.

Основные определения:

event loop (цикл событий) – программная конструкция, который ожидает прибытия и производит рассылку событий или сообщений в программе (см. паттерн Reactor);

coroutine (корутина, сопрограмма) – программная конструкция, способная вызываться и возвращать управление в цикл событий, сохраняя свое состояние между вызовами;

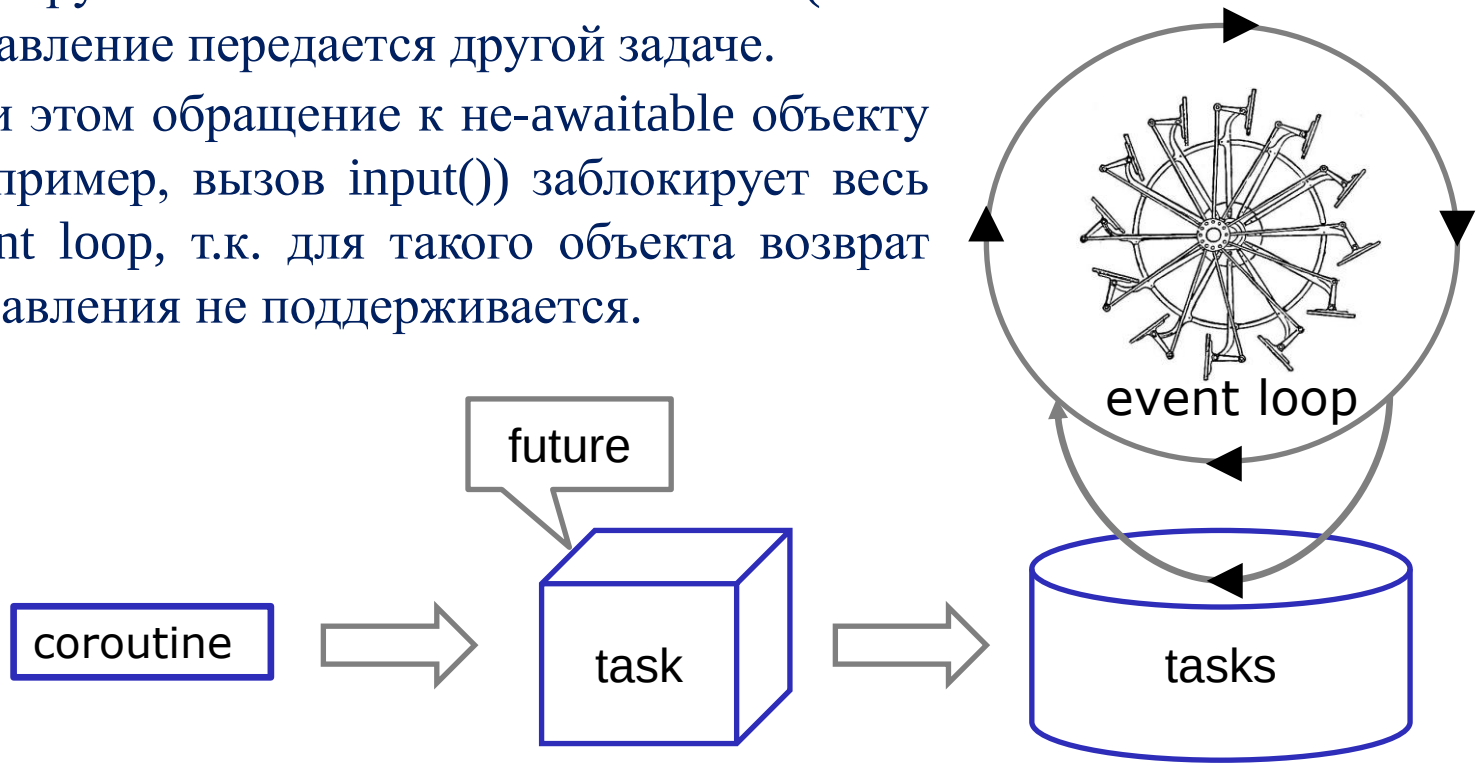
future (фьючерс) – низкоуровневый объект, хранящий текущий статус/результат асинхронной операции;

task (задача) – объект обработки цикла событий, инкапсулирующий работу с корутинами и фьючерсами.

# Asyncio (порядок работы)

При помощи **async def** мы создаем корутины (**native coroutines**, в отличие от генераторов), упаковываем их в задачи и передаем в event loop, который поочередно передает управление этим задачам. Как только задача блокируется на **awaitable** объекте (await вызов какой-либо корутины), управление передается другой задаче.

При этом обращение к не-awaitable объекту (например, вызов input()) заблокирует весь event loop, т.к. для такого объекта возврат управления не поддерживается.



# Asyncio (пример)

```
import asyncio

res = 0

# производим вычисления
async def calc(num):
    global res
    while num > 0:
        print(num)
        if num % 10 == 0:
            res = num
        num -= 1
        await asyncio.sleep(0.1)

# сообщаем о статусе вычислений
async def status():
    global res
    prev_res = 0
    while res > 0:
        if res != prev_res:
            print(f'{res} numbers remain')
            prev_res = res
        await asyncio.sleep(0.1)
```

# Asyncio (пример)

## Создание задач и запуск event loop:

```
if __name__ == '__main__':
    ev_loop = asyncio.get_event_loop()
    tasks = [ev_loop.create_task(calc(1000)),
             ev_loop.create_task(status())]
    futures = asyncio.wait(tasks)
    ev_loop.run_until_complete(futures)
    ev_loop.close()
```

## Альтернативный вариант:

```
async def main():
    tasks = [asyncio.create_task(calc(1000)),
             asyncio.create_task(status())]
    await asyncio.gather(*tasks)


if __name__ == '__main__':
    asyncio.run(main())
```

```
1000
1000 numbers remain
999
998
997
996
995
994
993
992
991
990
990 numbers remain
…
```

# Многопроцессность

Python позволяет не только запускать в одном процессе несколько потоков, но и создавать несколько дочерних процессов из основного процесса. При этом GIL здесь не используется в принципе. Попробуем посчитать скорость выполнения уже известной нам задачи с использованием нескольких процессов. Создание процессов с помощью модуля multiprocessing выполняется почти полностью аналогично созданию потоков с использованием threading.

# Многопроцессность

```
import multiprocessing
import time

def count(n):
    while n > 0:
        n -= 1

if __name__ == '__main__':  # обязательно для многопроцессного приложения
    c = 80000000
    start = time.perf_counter()
    p1 = multiprocessing.Process(target=count,args=(c,))
    p2 = multiprocessing.Process(target=count,args=(c,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print('{0:.2f} sec.'.format(time.perf_counter() - start))
```

```
7.96 sec.
```

Время выполнения почти в два раза быстрее, чем при использовании многопоточности, гринлетов или вообще без распараллеливания.

# Межпроцессное взаимодействие

При использовании нескольких процессов нужно решать проблему обмена данными между ними. Multiprocessing предлагает для этого два коммуникационных канала: Queue и Pipe. В обоих случаях данные должны быть сериализуемы (picklable). Очень большие блоки данных (более 32 MB, в зависимости от ОС) могут приводить к исключению ValueError.

# Межпроцессное взаимодействие: Queue

multiprocessing.Queue практически аналогична потокобезопасной очереди queue.Queue, но для нее не требуется ожидать окончания обработки объекта, переданного через очередь (и использовать методы .task_done и .join), т.к. передается копия объекта, изменение которой никак не влияет на оригинал.

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

```
[42, None, 'hello']
```

# Межпроцессное взаимодействие: Pipe

Pipe в отличие от Queue обеспечивает взаимодействие только двух процессов, представляя собой одно- или двунаправленный (по умолчанию) канал между ними. По производительности Pipe опережает Queue, тем более Queue состоит из каналов Pipe.

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send(['hello', 11, None])
    print('Client receives: {}'.format(conn.recv()))

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print('Server receives: {}'.format(parent_conn.recv()))
    parent_conn.send(['python', 3])
    p.join()
```

```
Server receives: ['hello', 11, None]
Client receives: ['python', 3]
```

# Синхронизация процессов

Для процессов также имеет место проблема синхронизации при доступе к разделяемым ресурсам (например, файлам). Для решения этой проблемы модуль multiprocessing предлагает все те же объекты синхронизации, что и threading, имеющие аналогичный интерфейс, но, естественно, другую внутреннюю реализацию.

```
from multiprocessing import Process, RLock

def f(lock, i):
    with lock:
        print('Process number {}'.format(i))

if __name__ == '__main__':
    lock = RLock()
    processes = []
    for num in range(5):
        p = Process(target=f, args=(lock, num))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()
```

```
Process number 0
Process number 1
Process number 4
Process number 2
Process number 3
```

# Использование разделяемой памяти

В параллельном программировании лучше стараться избегать совместно используемых данных и состояний настолько, насколько возможно. Если же избежать не получается, в модуле multiprocessing есть объекты Value и Array, обеспечивающие доступ к разделяемой памяти (shared memory) – еще одному способу межпроцессного взаимодействия. Разделяемая память - самый быстрый способ межпроцессного взаимодействия. Техника разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных вызовов ядра, а значит и без потерь производительности на переключение контекста между процессом и ядром. Сегмент разделяемой памяти подключается в свободную часть виртуального адресного пространства процесса. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.

# Использование разделяемой памяти

```
from multiprocessing import Process, Value, Array

def f(num, arr):
    num.value = 3.1415927
    for i in range(len(arr)):
        arr[i] = -arr[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))
    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()
    print('num={}'.format(num.value))
    print('arr={}'.format(arr[:]))
```

```
num=3.1415927
arr=[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Аргументы 'd' и 'i' в конструкторах Value и Array соответственно представляют собой коды типов: 'd' соответствует типу double (float двойной точности), а 'i' соответствует знаковому целому. Созданные объекты Value и Array являются процессо- и потокобезопасными.

# Создание пула процессов

Т.к. создание процессов в большинстве случаев происходит однотипно, модуль multithreading предоставляет объект Pool для создания сразу нескольких процессов, выполняющих одну функцию, но с разными аргументами.

```
This process 19992 processed values [1]
This process 15844 processed values [2]
This process 11320 processed values [3]
This process 16152 processed values [4]
This process 19992 processed values [1, 5]
This process 15844 processed values [2, 6]
[1, 8, 27, 64, 125, 216]
```

```
from multiprocessing import Pool
import time
import os

lst = []
def cube(x):
    lst.append(x)
    print(f'This process {os.getpid()} processed values {lst}')
    time.sleep(2)
    return x**3

if __name__ == '__main__':
    pool = Pool(processes=4)
    res = pool.map(cube, range(1,7))
    print(res)
```

# Практика

1. Написать функцию find_primes(end, start), которая ищет все простые числа в диапазоне от заданного числа start (по умолчанию 3) до заданного числа end. Далее необходимо:

   Запустить ее три раза последовательно в диапазоне от 3 до 10000, от 10001 до 20000, от 20001 до 30000.

   Запустить ее три раза с теми же аргументами, но каждый раз в отдельном потоке с помощью threading.Thread. Что будет, если 'забыть' выполнить start или join для потоков?

   Запустить ее три раза с теми же аргументами, но каждый раз в отдельном процессе с помощью multiprocessing.Process. Что будет, если 'забыть' выполнить start или join для процессов?

   Замерить время исполнения каждого варианта и сравнить результаты.
2. Реализовать запуск функции, осуществляющей операцию сложения для различных типов (integer, string, list) параллельно с различными наборами аргументов.

# Практика

3. * Создать несколько потоков таким образом, чтоб каждый из них мог хранить приватные данные, доступные только ему самому. Запустить потоки с одной функцией, выводящей в каждом потоке его имя приватные данные (имя исполняемого потока можно узнать, используя threading.current_thread().name).