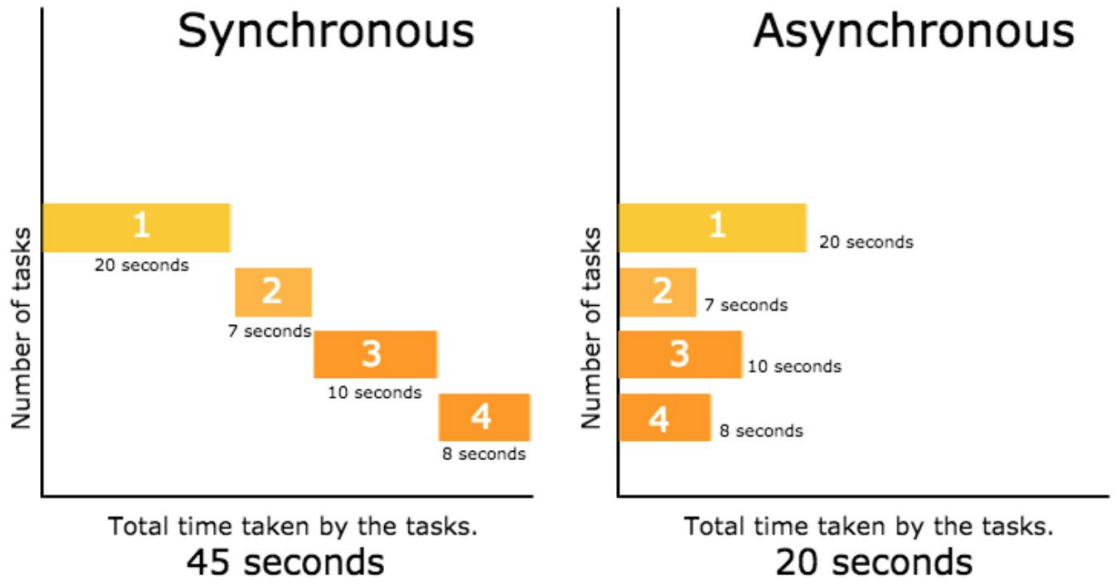
Основные понятия

синхрон или асинхрон, конкуренция или параллелизм

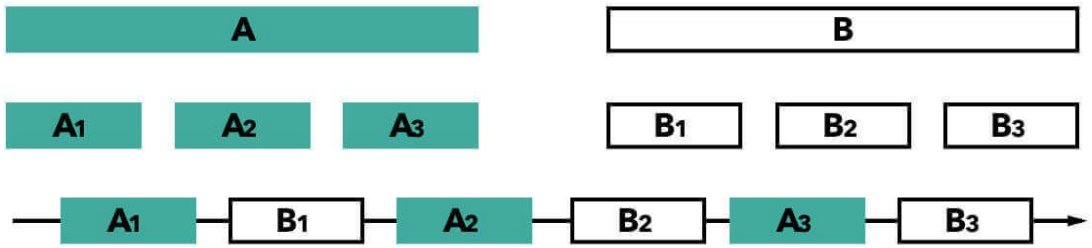
Как может исполняться код

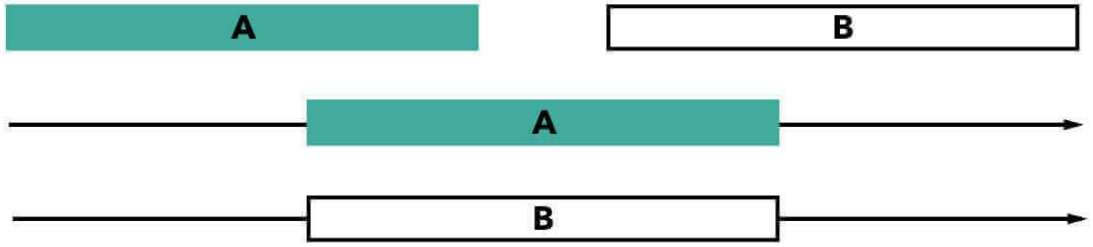
- **Синхронно** – выполнение программы подразумевает последовательное выполнение операций
- **Асинхронно** – предполагает возможность независимого выполнения задач. Асинхронные задачи не блокируют (не заставляют ждать завершения выполнения задачи) операции и обычно выполняются в фоновом режиме



Конкурентность и параллелизм

- **Конкурентность** — две или более задачи могут запускаться, выполняться и завершаться в перекрывающиеся периоды времени
- **Параллелизм** — исполнение нескольких задач одновременно. Для достижения параллелизма необходимо физическое одновременное исполнение зада





Конкурентность vs параллелизм

Конкурентность

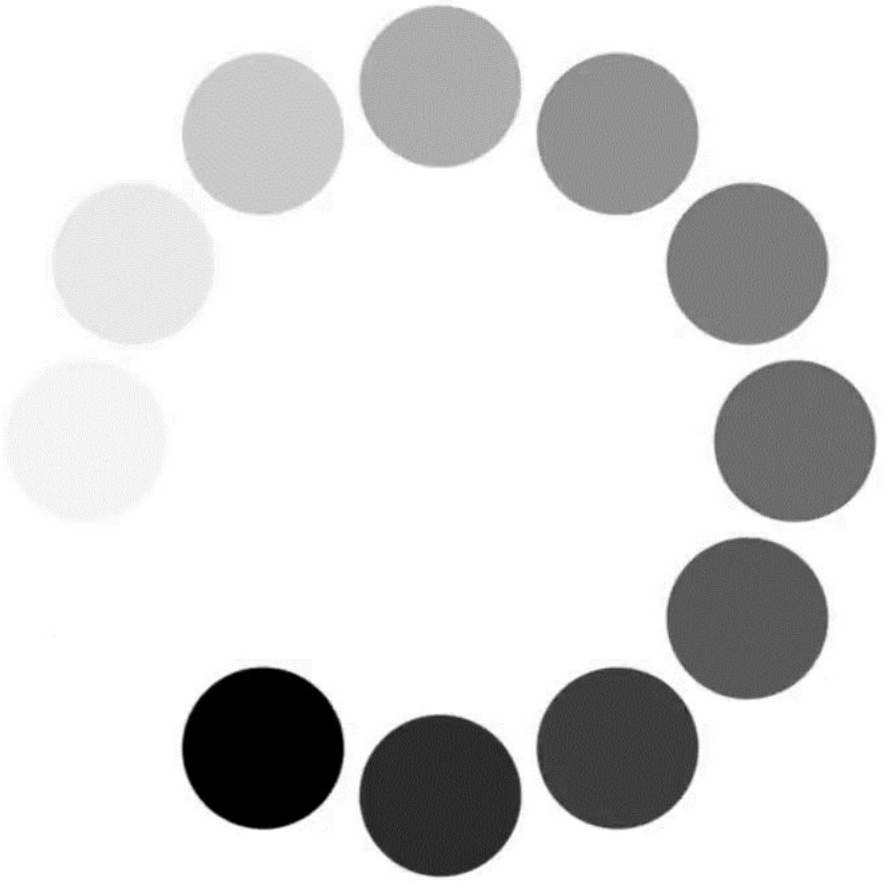
- Работаем с несколькими потоками управления
- Каждый поток управления получает квант времени на исполнение
- Каждый поток управления не обязательно исполняется до конца, прежде чем отдать управление следующему потоку

Параллелизм

- Необходима аппаратная составляющая (многоядерные/многопроцессорные системы)
- Наличие потоков управления не обязательно

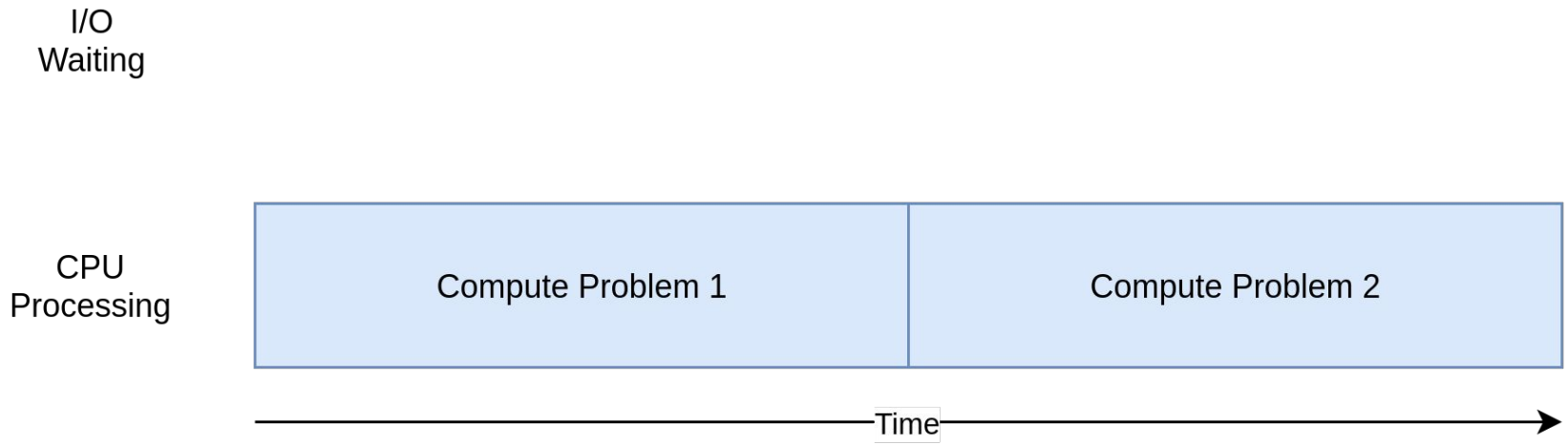
Когда мы ждем?

- **Ограничения ввода-вывода** (IO bound). Распространено шире других. Процессоры компьютеров безумно быстры – в сотни раз быстрее, чем компьютерные память, и в тысячи раз быстрее – чем диски или сети
- **Ограничения процессора** (CPU bound). Это случается, если выполняется большое количество объемных задач наподобие научных или графических расчетов



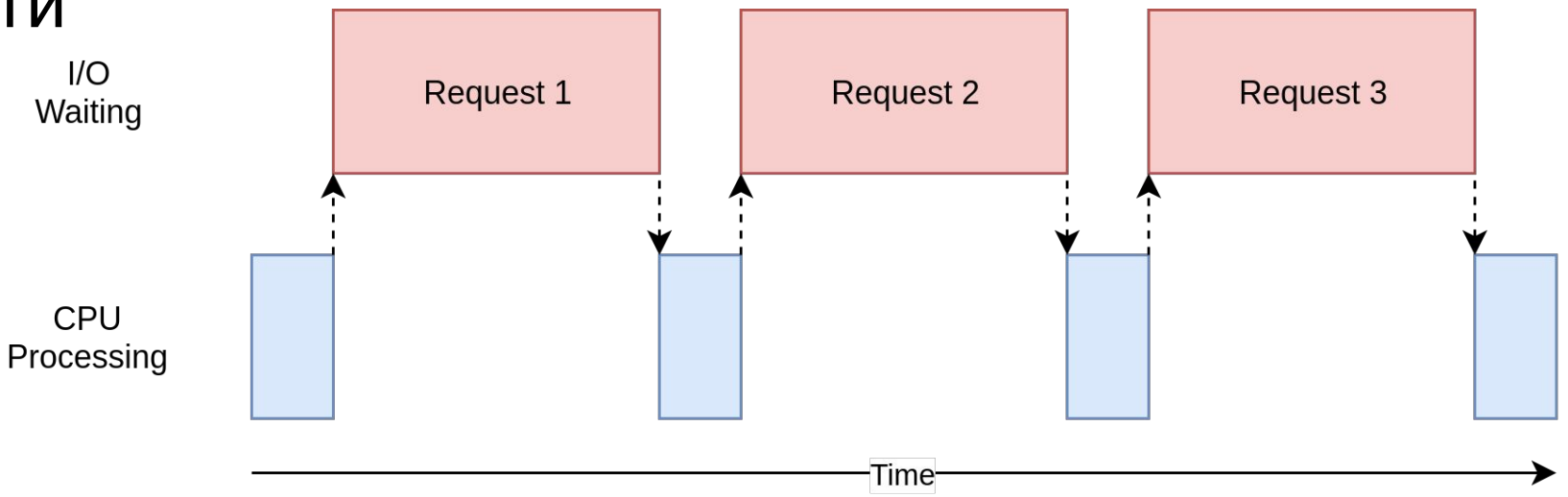
CPU-bound

CPU-bound – это когда скорость выполнения нашей программы напрямую зависит от того, насколько быстрый центральный процессор (частота ядер).



I/O-bound

I/O-bound – это когда скорость выполнения программы зависит от времени, которое мы тратим на ожидание выполнений операций ввода/вывода. Увеличение производительности ЦП не сказывается на производительности программы. На производительность сказывается скорость памяти, жесткого диска, сети



CPU-bound vs I/O-bound

CPU-bound

- Программа тратит много времени на выполнение CPU операций
- Для ускорения нужно найти способ выполнить больше вычислений в единицу времени

I/O-bound

- Программа тратит много времени на общение с медленными устройствами (сеть, жесткий диск и т.д.)
- Для ускорения нужно задействовать время ожидания для выполнения других задач

Нам нужно

При выполнение кода происходит передача управления из одной функции в другую при вызове и назад при окончании выполнения.

А нам нужна возможность передачи контроля управления программой, пока мы ожидаем.

Ваши вопросы

что необходимо прояснить в рамках данного раздела

threading

конкурентное выполнение в Python

Потоки

Поток - это способ позволить вашему компьютеру разбить отдельный процесс/программу на множество легковесных частей, которые выполняются параллельно.

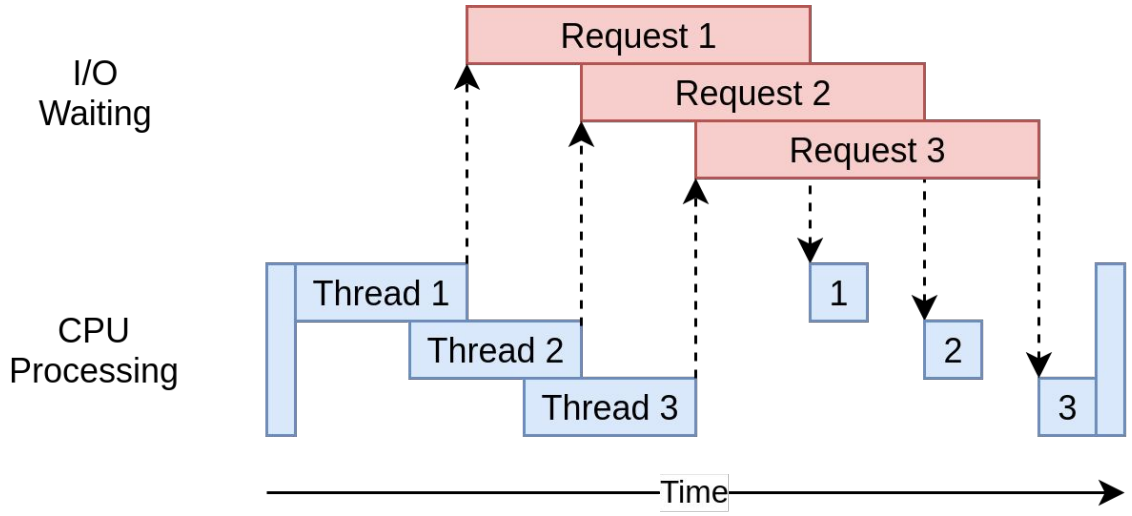
- Поток работает внутри процесса, имея доступ ко всему, что находится в процессе (общая память), что позволяет совместно выполнять некоторую задачу
- Нужно помнить о GIL. Все потоки будут выполняться на одном CPU, даже если задачи могут выполняться параллельно

```python
import threading


def doubler(number):
    """
    A function that can be used by a thread
    """
    print(threading.currentThread().getName() + '\n')
    print(number * 2)
    print()


if __name__ == '__main__':
    for i in range(5):
        my_thread = threading.Thread(target=doubler, args=(i,))
        my_thread.start()
```

Потоки

Так как у потоков общая память, то выполняя некоторую задачу, разбивая ее на потоки, мы должны обеспечить блокировки там, где это необходимо.

Механизм блокировок и разблокировок означает, что только один поток может писать в память в некоторый момент времени.



Как ОС распределяет выполнение потоков

ОС отслеживает все запущенные потоки, давая каждому из них немного времени, а затем переключаясь на другие, для того чтобы справедливо распределять работу и реагировать на действия пользователя.

```
>>> import sys
>>> sys.getswitchinterval()
0.005
```

Global Interpreter Lock (GIL)

GIL — это механизм блокировки, когда интерпретатор Python запускает в работу только один поток за раз. Т.е. только один поток может исполняться в байт-коде Python единовременно. GIL следит за тем, чтобы несколько потоков не выполнялись параллельно. GIL был представлен, чтобы сделать обработку памяти CPython проще и обеспечить наилучшую интеграцию с C(например, с расширениями).

- Одновременно может выполняться один поток
- Интерпретатор Python переключается между потоками для достижения конкурентности
- GIL применим к CPython. Но такие как, например, Jython и IronPython не имеют GIL
- GIL делает однопоточные программы быстрыми
- Операциям ввода/вывода GIL обычно не мешает (мешает CPU-bound)
- GIL позволяет легко интегрировать непотокобезопасные библиотеки на C, благодаря GIL у нас есть много высокопроизводительных расширений/модулей, написанных на C
- (CPython < 3.2) Отпускается каждые N выполненных инструкций байт-кода
- (CPython >= 3.2) Отпускается каждые N секунд

Как обойти GIL

GIL необходим, потому что управление памятью CPython (реализация Python по умолчанию) не является потокобезопасным. Из-за этого ограничения потоки в Python являются конкурентными, но не параллельными. Чтобы обойти это, в Python есть отдельный модуль multiprocessing, не ограниченный GIL, который запускает отдельные процессы, обеспечивая параллельное выполнение вашего кода. Использование модуля multiprocessing почти идентично использованию модуля threading.

Осторожно

Потоки опасны. Как и управление памятью в языках C и C++ они могут вызвать появление ошибок, которые ужасно трудно найти и исправить. Для того, чтобы использовать потоки, весь код программы – и код внешних библиотек, которые он использует, - должен быть потокобезопасным

Как сделать безопасно

Классический способ разделить данные безопасно – **разместить** программную **блокировку перед изменением переменной** в потоке. Это позволит оградить ее значение от других потоков и внести свои изменения. Блокировки довольно сложно организовать правильно.

Глобальные данные

Потоки могут быть полезны и безопасны, когда речь не идет о глобальных данных. В частности, потоки полезно использовать для экономии времени при ожидании завершения некоторой операции ввода/вывода. В этих случаях не придется сражаться за данные, поскольку у каждого из низ них имеется свой набор переменных.

Потоки могут иметь и глобальные данные по хорошей причине. Фактически самая распространенная причина использования нескольких потоков – это возможность разделить между ними работу над некоторыми данными, поэтому можно ожидать, что некоторые данные будет изменены.

Блокировки, Lock-объект

Lock-объект может находится в двух состояниях:

- захваченное (заблокированное)
- не захваченное (не заблокированное, свободное)

Для работы с Lock-объектом используются методы acquire() и release(). Если Lock свободен, то вызов метода acquire() переводит его в заблокированное состояние. Повторный вызов acquire() приведет к блокировке инициировавшего это действие потока до тех пор, пока Lock не будет разблокирован каким-то другим потоком с помощью метода release(). Вывоз метода release() на свободном Lock-объекте приведет к выбросу исключения RuntimeError.

Освободить Lock-объект может любой поток (на обязательно тот, который вызвал acquire()).

```python
from threading import Thread, Lock


lock = Lock()


lock_obj.acquire()
try:
    # Работа с разделяемым ресурсом
finally:
    lock_obj.release()


with lock_obj:
    # Работа с разделяемым ресурсом
```

Блокировки, RLock-объект

В отличии от рассмотренного выше Lock-объекта RLock может освободить только тот поток, который его захватил. Повторный захват потоком уже захваченного RLock-объекта не блокирует его. RLock-объекты поддерживают возможность вложенного захвата, при этом освобождение происходит только после того, как был выполнен release() для внешнего acquire(). Сигнатуры и назначение методов release() и acquire() RLock-объектов совпадают с приведенными для Lock, но в отличии от него у RLock нет метода locked(). RLock-объекты поддерживают протокол менеджера контекста.

```python
import threading
import time

data = []
lock = threading.RLock()


def who_am_i(what):
    print(f"Thread {threading.current_thread()} says: {what}")
    lock.acquire()
    global data
    data.append(what)
    time.sleep(1)
    lock.release()


if __name__ == '__main__':
    for n in range(4):
        thread = threading.Thread(
            target=who_am_i,
            args=(f"I'm function {n}",)
        )
        thread.start()
    print(data)
```

Условные переменные (threading.Condition)

Основное назначение условных переменных – это синхронизация работы потоков, которая предполагает ожидание готовности некоторого ресурса и оповещение об этом событии. Наиболее явно такой тип работы выражен в паттерне Producer-Consumer (Производитель – Потребитель). Условные переменные для организации работы внутри себя используют Lock- или RLock-объекты, захватом и освобождением которых управлять не придется, хотя и возможно, если возникнет такая необходимость.

Порядок работы с условными переменными выглядит так:

- На стороне Producer'а: произвести работы по подготовке ресурса, после того, как ресурс готов оповестить об этом ожидающие потоки с помощью методов notify() или notify_all(). Разница между ними в том, что notify() разблокирует только один поток (если он вызван без параметров), а notify_all() все потоки, которые находятся в режиме ожидания.
- На стороне Consumer'а: проверить доступен ли ресурс, если нет, то перейти в режим ожидания с помощью метода wait(), и ожидать оповещение от Producer'а о том, что ресурс готов и с ним можно работать. Метод wait() может быть вызван с таймаутом, по истечении которого поток выйдет из состояния блокировки и продолжит работу.

```python
from threading import Condition, Thread
from queue import Queue
from time import sleep

cv = Condition()
q = Queue()


# Consumer function for order processing
def order_processor(name):
    while True:
        with cv:
            # Wait while queue is empty
            while q.empty():
                cv.wait()
            try:
                # Get data (order) from queue
                order = q.get_nowait()
                print(f"{name}: {order}")
                # If get "stop" message then stop thread
                if order == "stop":
                    break
            except:
                pass
            sleep(0.1)


# Run order processors
Thread(target=order_processor, args=("thread 1",)).start()
Thread(target=order_processor, args=("thread 2",)).start()
Thread(target=order_processor, args=("thread 3",)).start()
# Put data into queue
for i in range(10):
    q.put(f"order {i}")
# Put stop-commands for consumers
for _ in range(3):
    q.put("stop")
# Notify all consumers
with cv:
    cv.notify_all()
```

Условные переменные (threading.Condition)

В этом примере мы создаем функцию order_processor, которая может реализовывать в себе бизнес логику, например, обработку заказа. При этом, если она получает сообщение stop, то прекращает свое выполнение. В главном потоке мы создаем и запускаем три потока для обработки заказов. Запущенные потоки видят, что очередь пуста и “встают на блокировку” при вызове wait(). В главном потоке в очередь добавляются десять заказов и сообщения для остановки обработчиков, после этого вызывается метод notify_all() для оповещения всех заблокированных потоков о том, что данные для обработки есть в очереди.

При создании объекта Condition вы можете передать в конструктор объект Lock или RLock, с которым хотите работать. Перечислим методы объекта Condition с кратким описанием:

- acquire(*args) – захват объекта-блокировки
- release() – освобождение объекта-блокировки
- wait(timeout=None) – блокировка выполнения потока до оповещения о снятии блокировки. Через параметр timeout можно задать время ожидания оповещения о снятии блокировки. Если вызвать wait() на Условной переменной, у которой предварительно не был вызван acquire(), то будет выброшено исключение RuntimeError
- wait_for(predicate, timeout=None) – метод позволяет сократить количество кода, которое нужно написать для контроля готовности ресурса и ожидания оповещения
- notify(n=1) – снимает блокировку с остановленного методом wait() потока. Если необходимо разблокировать несколько потоков, то для этого следует передать их количество через аргумент n
- notify_all() – снимает блокировку со всех остановленных методом wait() потоков.

```python
from threading import Condition, Thread
from queue import Queue
from time import sleep

cv = Condition()
q = Queue()


# Consumer function for order processing
def order_processor(name):
    while True:
        with cv:
            # Wait while queue is empty
            while q.empty():
                cv.wait()
            try:
                # Get data (order) from queue
                order = q.get_nowait()
                print(f"{name}: {order}")
                # If get "stop" message then stop thread
                if order == "stop":
                    break
            except:
                pass
            sleep(0.1)


# Run order processors
Thread(target=order_processor, args=("thread 1",)).start()
Thread(target=order_processor, args=("thread 2",)).start()
Thread(target=order_processor, args=("thread 3",)).start()
# Put data into queue
for i in range(10):
    q.put(f"order {i}")
# Put stop-commands for consumers
for _ in range(3):
    q.put("stop")
# Notify all consumers
with cv:
    cv.notify_all()
```

Семафоры (threading.Semaphore)

Реализация классического семафора, предложенного Дейкстрой. При каждом вызове метода acquire() происходит уменьшение счетчика семафора на единицу, а при вызове release() – увеличение. Значение счетчика не может быть меньше нуля, если на момент вызова acquire() его значение равно нулю, то происходит блокировка потока до тех пор, пока не будет вызван release().

Семафоры поддерживают протокол менеджера контекста.

Для работы с семафорами в Python есть класс Semaphore, при создании его объекта можно указать начальное значение счетчика через параметр value. Semaphore предоставляет два метода:

- acquire(blocking=True, timeout=None). Если значение внутреннего счетчика больше нуля, то счетчик уменьшается на единицу и метод возвращает True. Если значение счетчика равно нулю, то вызвавший данный метод поток блокируется, до тех пор, пока не будет кем-то вызван метод release(). Дополнительно при вызове метода можно указать параметры blocking и timeout, их назначение совпадает с acquire() для Lock
- release(). Увеличивает значение внутреннего счетчика на единицу.

Существует ещё один класс, реализующий алгоритм семафора BoundedSemaphore, в отличии от Semaphore, он проверяет, чтобы значение внутреннего счетчика было не больше того, что передано при создании объекта через аргумент value, если это происходит, то выбрасывается исключение ValueError.

С помощью семафоров удобно управлять доступом к ресурсу, который имеет ограничение на количество одновременных обращений к нему (например, количество подключений к базе данных и т.п.).

В качестве примера приведем программу, моделирующую продажу билетов: обслуживание одного клиента занимает одну секунду, касс всего три, клиентов пять.

```python
from threading import Thread, BoundedSemaphore
from time import sleep, time

ticket_office = BoundedSemaphore(value=3)


def ticket_buyer(number):
    start_service = time()
    with ticket_office:
        sleep(1)
        print(f"client {number}, service time: {time() - start_service}")


buyer = [Thread(target=ticket_buyer, args=(i,)) for i in range(5)]
for b in buyer:
    b.start()
```

События (threading.Event)

События по своему назначению и алгоритму работы похожи на рассмотренные ранее условные переменные. Основная задача, которую они решают – это взаимодействие между потоками через механизм оповещения. Объект класса Event управляет внутренним флагом, который сбрасывается с помощью метода clear() и устанавливается методом set(). Потоки, которые используют объект Event для синхронизации блокируются при вызове метода wait(), если флаг сброшен.

Методы класса Event:

- is_set(). Возвращает True если флаг находится в взведенном состоянии
- set(). Переводит флаг в взведенное состояние
- clear(). Переводит флаг в сброшенное состояние
- wait(timeout=None). Блокирует вызвавший данный метод поток если флаг соответствующего Event-объекта находится в сброшенном состоянии. Время нахождения в состоянии блокировки можно задать через параметр timeout

```python
from threading import Thread, Event

event = Event()


def worker(name: str):
    event.wait()
    print(f"Worker: {name}")


# Clear event
event.clear()
# Create and start workers
workers = [Thread(target=worker, args=(f"wrk {i}",)) for i in range(5)]
for w in workers:
    w.start()
print("Main thread")
event.set()
```

Таймеры (threading.Timer)

Модуль threading предоставляет удобный инструмент для запуска задач по таймеру – класс Timer.

При создании таймера указывается функция, которая будет выполнена, когда он сработает. Timer реализован как поток, является наследником от Thread, поэтому для его запуска необходимо вызвать start(), если необходимо остановить работу таймера, то вызовите cancel().

Конструктор класса Timer:

- Timer(interval, function, args=None, kwargs=None)

Параметры:

- Interval. Количество секунд, по истечении которых будет вызвана функция function.
- function. Функция, вызов которой нужно осуществить по таймеру.
- args, kwargs. Аргументы функции function.

Методы класса Timer:

- cancel(). Останавливает выполнение таймера

```
1  from threading import Timer
2
3  timer = Timer(interval=3,function=lambda: print("Message from Timer!"))
4  timer.start()
```

Барьеры (threading.Barrier)

Последний инструмент для синхронизации работы потоков, который мы рассмотрим является Barrier. Он позволяет реализовать алгоритм, когда необходимо дождаться завершения работы группы потоков, прежде чем продолжить выполнение задачи.

Конструктор класса:

- Barrier(parties, action=None, timeout=None)

Параметры:

- parties. Количество потоков, которые будут работать в рамках барьера.
- action. Определяет функцию, которая будет вызвана, когда потоки будут освобождены (достигнут барьера).
- timeout. Таймаут, который будет использовать как значение по умолчанию для методов wait().

Свойства и методы класса:

- wait(timeout=None). Блокирует работу потока до тех пор, пока не будет получено уведомление либо не пройдет время указанное в timeout.
- reset(). Переводит Barrier в исходное (пустое) состояние. Потокам, ожидающим уведомления, будет передано исключение BrokenBarrierError.
- abort(). Останавливает работу барьера, переводит его в состояние “разрушен” (broken). Все текущие и последующие вызовы метода wait() будут завершены с ошибкой с выбросом исключения BrokenBarrierError.
- parties. Количество потоков, которое нужно для достижения барьера.
- n_waiting. Количество потоков, которое ожидает срабатывания барьера.
- broken. Значение флага равное True указывает на то, что барьер находится в “разрушенном” состоянии. Данный объект синхронизации может применяться в случаях когда необходимо дождаться результатов работы всех потоков (как в примере выше), либо для синхронизации процесса инициализации потоков, когда перед стартом их работы требуется, чтобы все потоки выполнили процедуру инициализации.

```python
from threading import Barrier, Thread
from time import sleep

br = Barrier(3)
store = []


def f1(x):
    print("Calc part1")
    store.append(x ** 2)
    sleep(0.5)
    br.wait()


def f2(x):
    print("Calc part2")
    store.append(x * 2)
    sleep(1)
    br.wait()


Thread(target=f1, args=(3,)).start()
Thread(target=f2, args=(7,)).start()

br.wait()

print("Result: ", sum(store))
```

Ждем пока thread завершит работу

Если необходимо дождаться завершения работы потока(ов) перед тем как начать выполнять какую-то другую работу, то воспользуйтесь методом join().

```python
from threading import Thread
from time import sleep


def func():
    for i in range(5):
        print(f"from child thread: {i}")
        sleep(0.5)


th = Thread(target=func)
th.start()
for i in range(5):
    print(f"from main thread: {i}")
    sleep(1)

th1 = Thread(target=func)
th2 = Thread(target=func)
th1.start()
th2.start()
th1.join()
th2.join()
print("--> stop")
```

Как можно завершить thread

В Python у объектов класса Thread нет методов для принудительного завершения работы потока. Один из вариантов решения этой задачи – это создать специальный флаг, через который потоку будет передаваться сигнал остановки. Доступ к такому флагу должен управляться объектом синхронизации.

```python
from threading import Thread, Lock
from time import sleep

lock = Lock()
stop_thread = False


def infinite_worker():
    print("Start infinite_worker()")
    while True:
        print("--> thread work")
        lock.acquire()

        if stop_thread is True:
            break
        lock.release()

        sleep(0.1)
    print("Stop infinite_worker()")


thread = Thread(target=infinite_worker)
thread.start()
sleep(2)

# Stop thread
lock.acquire()
stop_thread = True
lock.release()
```

Потоки-демоны

Есть такая разновидность потоков, которые называются демоны. Python-приложение не будет закрыто до тех пор, пока в нем работает хотя бы один недемонический поток.

```python
from threading import Thread
from time import sleep


def func():
    for i in range(5):
        print(f"from child thread: {i}")
        sleep(0.5)


thread = Thread(target=func)
thread.start()
print("App stop")
thread.join()

print('\n')
thread = Thread(target=func, daemon=True)
thread.start()
print("App stop")
```

Ваши вопросы

что необходимо прояснить в рамках данного раздела

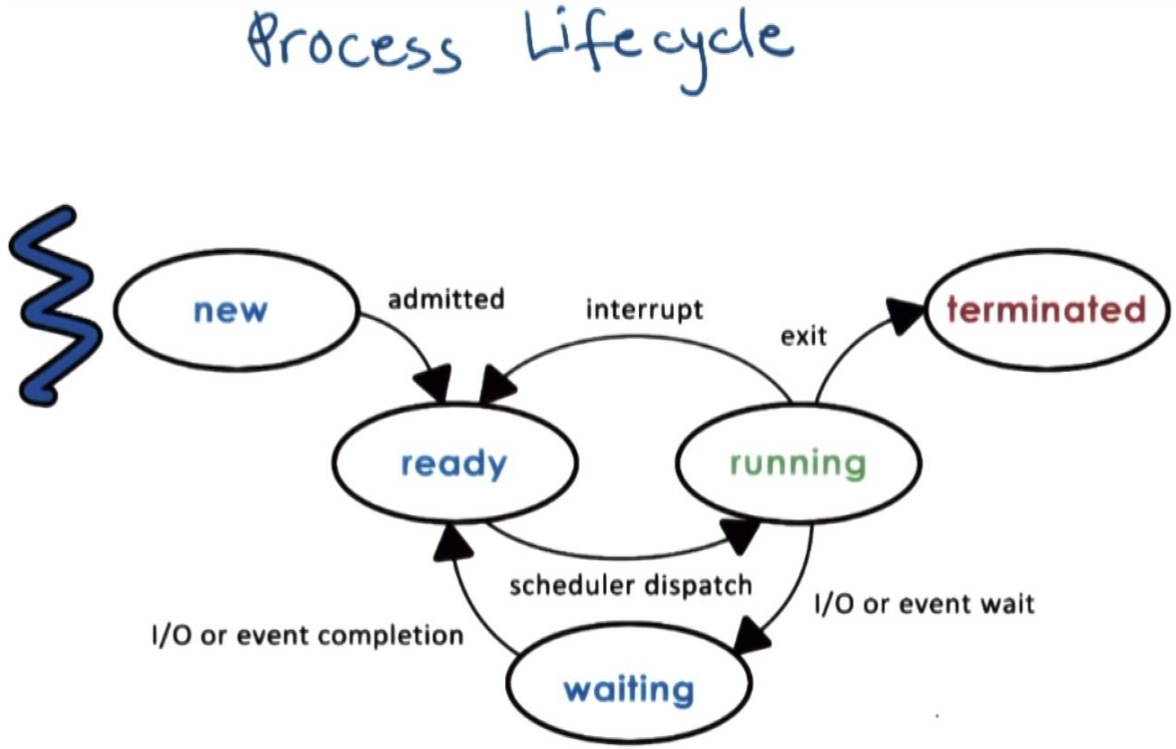
multiprocessing

параллельное выполнение программного кода

Что такое процессы

Когда вы запускаете отдельную программу, ваша система создает один процесс. Он использует системные ресурсы (ЦП, ОЗУ, Место на диске) и структуры данных в ядре ОС (файлы и сетевые соединения, статистика использования и т. д.).

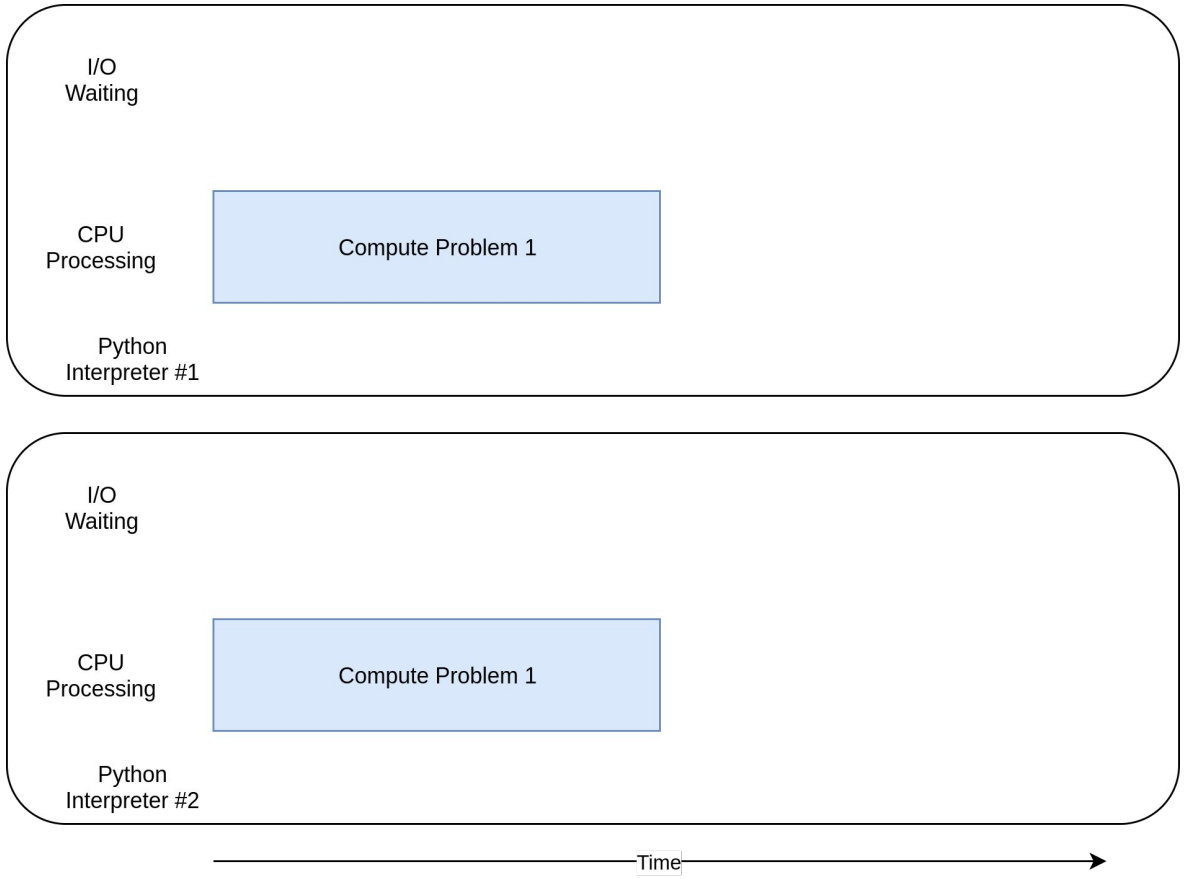
Процесс изолирован от других процессов – он не может видеть, что делают другие процессы, или мешать им.



Процессы

Процессы, как и потоки, всегда исполняются конкурентно, но могут исполняться еще и параллельно, в зависимости от наличия аппаратной составляющей.

- GIL не ограничивает параллельное выполнение



multiprocessing

Вы можете запустить функцию Python как отдельны процесс или даже как несколько независимых процессов с помощью модуля multiproccesing. Так как все процессы независимы, то они не делят память.

- Позволяет утилизировать все ядра CPU

```python
import multiprocessing
import os


def do_this(what):
    who_am_i(what)


def who_am_i(what):
    print(f"Process {os.getpid()} says: {what}")


if __name__ == '__main__':
    who_am_i("I'm the main program")
    for n in range(4):
        process = multiprocessing.Process(
            target=do_this,
            args=(f"I'm function {n}",)
        )
        process.start()
```

«Убиваем» процесс

В отличии от потоков, работу процессов можно принудительно завершить, для этого класс Process предоставляет набор методов:

- **terminate()** – принудительно завершает работу процесса. В Unix отправляется команда SIGTERM, в Windows используется функция TerminateProcess()
- **kill()** – метод аналогичный terminate() по функционалу, только вместо SIGTERM в Unix будет отправлена команда SIGKILL

```python
import multiprocessing
import time
import os


def who_am_i(name):
    print(f"I'm {name}, in process {os.getpid()}")


def loopy(name):
    who_am_i(name)
    start = 1
    stop = 1000000
    for num in range(start, stop):
        print(f"\tNumber of {num} of {stop}. Honk!")
        time.sleep(1)


if __name__ == '__main__':
    who_am_i("I'm the main program")
    process = multiprocessing.Process(
        target=loopy,
        args=('loopy',)
    )
    process.start()
    for i in range(5):
        print('hi')
        time.sleep(1)
    process.terminate()
```

Потоки vs процессы

Потоки

- Ограничены GIL
- Время создания меньше
- Потребление памяти меньше
- Время коммуникации меньше
- Общий контекст (простая коммуникация между потоками)
- Проблемы с блокировками и синхронизацией
- Утилизируют одно ядро (Конкурентное выполнение)
- За переключение задач отвечает ОС
- Сложно писать и поддерживать код в надлежащем состоянии

Процессы

- Не ограничены GIL
- Время создания больше
- Потребление памяти больше
- Время коммуникации больше
- Независимость друг от друга (с Python 3.8 есть shared memory)
- Утилизирует много ядер (обход GIL, хорошо для CPU bound)
- Можно «убить» процесс

Ваши вопросы

что необходимо прояснить в рамках данного раздела

asyncio

корутины, event loop, async-await

asyncio

asyncio – библиотека для написания конкурентного кода с помощью синтаксиса async/await

- asyncio работает в рамках одного процесса и одного потока (в отличие от многопоточности). За переключение между задачами отвечает event loop (для остановки и возобновления задач используются корутины и генераторы)

```python
import asyncio


async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')


# Python 3.4+
event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(main())
event_loop.close()

# Python 3.7+
asyncio.run(main())
```

Вспомним генераторы

Генератор – функция, которая может приостанавливать и возобновлять свое выполнение (передавать контроль выполнения)

```python
from time import sleep


def print_counter():
    counter = 0
    while True:
        print(counter)
        yield
        counter += 1


def print_hi():
    counter = 0
    while True:
        if counter % 3 == 0:
            print('Hi from second generator')
        counter += 1
        yield


def main(queue):
    while True:
        gen = queue.pop(0)
        next(gen)
        queue.append(gen)
        sleep(0.5)


if __name__ == '__main__':
    gen_1 = print_counter()
    gen_2 = print_hi()
    gen_queue = [gen_1, gen_2]
    main(gen_queue)
```

Событийный цикл (Event loop)

Цикл событий - это процесс, который ожидает срабатывания некоторых триггеров, а затем выполняет определенные (запрограммированные) действия, как только эти триггеры срабатывают. Триггеры часто возвращают какое-то объект, будь то «promise» (синтаксис JavaScript) или «future» (синтаксис Python) как признак того, что задача была добавлена. После завершения задачи future возвращает значение, переданное обратно из вызываемой функции (при условии, что функция действительно возвращает значение).

- Передача функции в качестве одного из параметров другой функции и как следствие выполнение некоторой функции в ответ на вызов другой называется «обратным вызовом» (callback)

```python
import asyncio


async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')


# Python 3.4+
event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(main())
event_loop.close()

# Python 3.7+
asyncio.run(main())
```

Событийный цикл (Event loop)

Цикл событий (event loop) отслеживает события ввода/вывода и переключает задачи, которые готовы и ждут операции ввода/вывода.

Есть функции, которые выполняют асинхронные операции ввода-вывода. Мы передаем свои функции циклу событий и просим его запустить их для нас. Цикл событий возвращает нам объект Future, словно обещание, что в будущем мы что-то получим. Мы держимся за обещание, время от времени проверяем, имеет ли оно значение, и, наконец, когда значение получено, мы используем его в некоторых других операциях.

Корутины

Корутины — специальные функции, похожие на генераторы python, от которых ожидают (await), что они будут отдавать управление обратно в цикл событий. Необходимо, чтобы они были запущены именно через цикл событий.

Корутина может быть вызвана только из корутины или из событийного цикла.

Внутри корутины можно вызывать синхронные функции.

```python
import asyncio


async def some_coroutine():
    await asyncio.sleep(2)
    print('Hi From coroutine!')
```

Футуры (Future)

Футуры — объекты, в которых хранится текущий результат выполнения какой-либо задачи. Это может быть информация о том, что задача ещё не обработана или уже полученный результат; а может быть вообще исключение

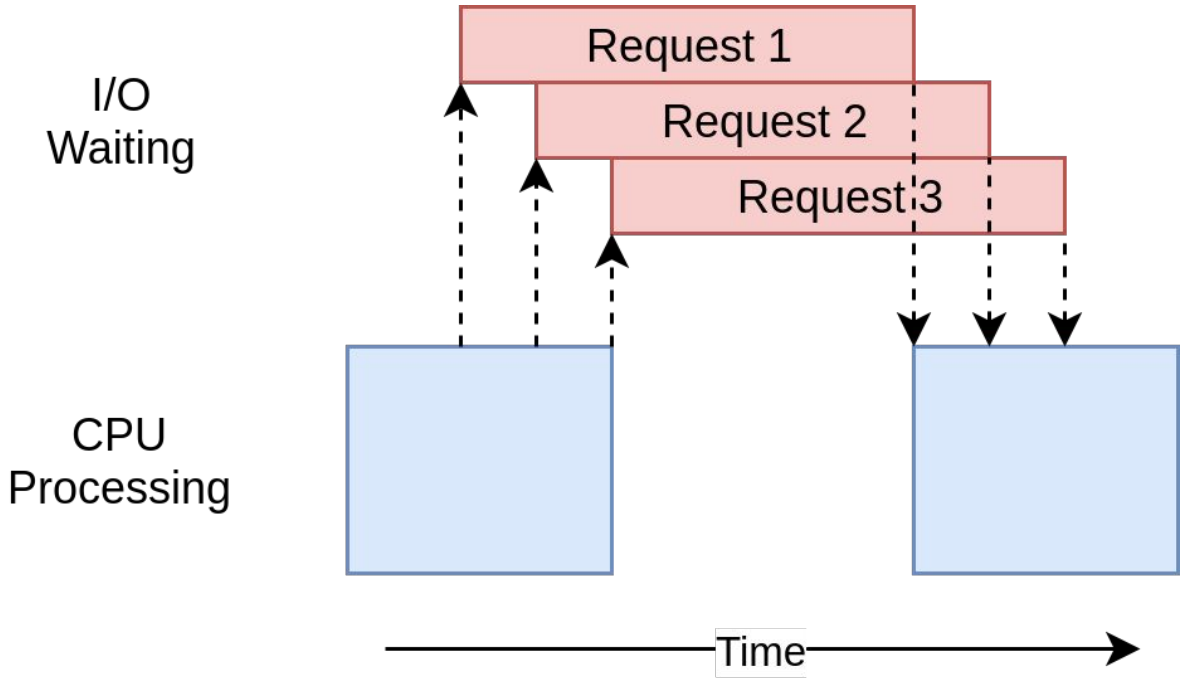
Состояние Future

- ожидание (pending)
- выполнение (running)
- выполнено (done)
- отменено (cancelled)

Когда футура находится в состояние done, у неё можно получить результат выполнения. В состояниях pending и running такая операция приведёт к исключению InvalidStateError, а в случае canelled будет CancelledError, и наконец, если исключение произошло в самой корутине, оно будет сгенерировано снова (также, как это сделано при вызове exception).

asyncio

- Быстрее, чем потоки
- Хорошо для I/O-bound
- Колбеки заменены на await
- Нужно использовать неблокирующие версии библиотек для адекватной работы (aiohttp, aioredis, aiofile)
- Тяжелее писать, чем потоки, но более предсказуемы и управляемы
- Конкурентное выполнение на одном ядре CPU
- Передача управления между корутинами осуществляется явно при вызове оператора await
- Если смешать multiprocessing и asyncio, то можно задействовать все ядра процессора



Python 3.4+

- Для объявления корутин использовался декоратор @asyncio.coroutine (deprecated, удаляется в Python 3.10)
- Передача управления с помощью yield from

```python
import asyncio


@asyncio.coroutine
def counter():
    count = 0
    while True:
        print(count)
        count += 1
        yield from asyncio.sleep(1)


@asyncio.coroutine
def print_time():
    count = 0
    while True:
        if count % 3 == 0:
            print(f"{count} секунд прошло")
        count += 1
        yield from asyncio.sleep(1)


@asyncio.coroutine
def main():
    task_1 = asyncio.ensure_future(print_time())
    task_2 = asyncio.ensure_future(counter())
    yield from asyncio.gather(task_1, task_2)


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
    loop.close()
```

Python 3.5+

- Для объявления корутин используется async def. Чтобы вызвать асинхронную функцию, вы должны либо использовать ключевое слово await из другой асинхронной функции, либо вызвать create_task() непосредственно из цикла событий, который можно получить из asyncio вызвав get_event_loop()
- Вместо yield from используется await
- async with позволяет ожидать асинхронных ответов и файловых операций
- async for выполняет итерацию по асинхронному потоку

```python
import asyncio


async def counter():
    count = 0
    while True:
        print(count)
        count += 1
        await asyncio.sleep(1)


async def print_time():
    count = 0
    while True:
        if count % 3 == 0:
            print(f"{count} секунд прошло")
        count += 1
        await asyncio.sleep(1)


async def main():
    task_1 = asyncio.create_task(print_time())
    task_2 = asyncio.create_task(counter())
    await asyncio.gather(task_1, task_2)


if __name__ == '__main__':
    # loop = asyncio.get_event_loop()
    # loop.run_until_complete(main())
    # loop.close()
    # Python 3.7+
    asyncio.run(main())
```

Тестим синхрон vs асинхрон

```python
import requests
from time import time


URL = 'https://loremflickr.com/320/240'


def get_file():
    response = requests.get(URL)
    return response


def write_file(response):
    filename = response.url.split('/')[-1]
    with open('pic/' + filename, 'wb') as file:
        file.write(response.content)


def main():
    t0 = time()
    for i in range(10):
        write_file(get_file())
    print(time() - t0)


if __name__ == '__main__':
    main()
```

```python
import asyncio
from time import time

import aiofiles
import aiohttp


URL = 'https://loremflickr.com/320/240'


async def get_file(session: aiohttp.ClientSession):
    async with session.get(URL, allow_redirects=True) as response:
        data = await response.read()
        await write_file(data)


async def write_file(data):
    filename = f'file-{int(time() * 1000)}.jpeg'
    async with aiofiles.open('pic/' + filename, 'wb') as file:
        await file.write(data)


async def main():
    t0 = time()
    tasks = []
    async with aiohttp.ClientSession() as session:
        for i in range(10):
            tasks.append(asyncio.create_task(get_file(session)))
        await asyncio.gather(*tasks)
    print(time() - t0)
    await asyncio.sleep(0.5)


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

Запуск задач конкурентно

Для конкурентного запуска задач – их необходимо передать в метод gather()

```python
import asyncio


async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")


async def main():
    # Schedule three calls *concurrently*:
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )


asyncio.run(main())
```

Асинхронный менеджер контекста

Асинхронный менеджер контекста работает также, как и синхронный, за исключением того, что методы называются __aenter__ и __aexit__, а также вызов не with, а async with

```python
import asyncio


class AsyncContextManager:
    async def __aenter__(self):
        print('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        print('exiting context')


async def main():
    async with AsyncContextManager() as ctx:
        await asyncio.sleep(2)


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

Асинхронный итератор

Создание асинхронного итератора аналогично синхронному.

Метод __aiter__ не корутина!

```python
import asyncio


class AsyncIterator:
    def __aiter__(self):
        return self

    async def __anext__(self):
        return 123


async def main():
    async for i in AsyncIterator():
        print(i)
        break


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

Тестирование корутин

Тестировать асинхронные функции с помощью pytest так же просто, как тестировать синхронные функции. Просто установите пакет pytest-asyncio с помощью pip, пометьте свои тесты ключевым словом async и примените декоратор, который сообщает pytest об асинхронности: @pytest.mark.asyncio.

В качестве альтернативы pytest-asyncio, можно создать pytest фикстуру которая генерирует цикл событий asyncio:

```
1  @pytest.fixture
2  def event_loop():
3      loop = asyncio.get_event_loop()
4      yield loop
```

Резюмируя вышесказанное

Тип конкурентности	Особенности работы	Критерий использования	Пример
multiprocessing	Много процессов, высокая утилизация CPU, параллельное выполнение	CPU-bound	У нас 10 кухонь, 10 поваров, 10 блюд для приготовления
threading	Один процесс, много потоков, конкурентное выполнение, ОС отвечает за переключение потоков	Быстрый I/O, ограниченное количество подключений	1 кухня, 10 поваров, 10 блюд для приготовления. Кухня переполнена, когда все повара работают вместе
asyncio	Один процесс, один поток, конкурентное выполнение, за переключение задач отвечает event loop (задачи передают контроль управления)	Медленный I/O, много подключений	1 кухня, 1 повар, 10 блюд для приготовления

Вопрос-ответ

? Так что же мне использовать?

✔ Если у нас зависимость от CPU – multiprocessing, если I/O, то если можем, то asyncio, если нет, то threading.

✔ Для I/O-bound проблем в Python community сложилось следующее правило: “Используй asyncio, когда ты может, threading, когда ты должен”. Когда вы пишете новый код, используйте asyncio. Если вам нужно взаимодействовать со старыми библиотеками или теми, которые не поддерживают asyncio, вам может быть лучше использовать threading

```
if io_bound:
    if py35_plus_only and vary_slow_IO:
        print("asyncio")
    else:
       print("threads")
else:
    print("multiprocessing")
```