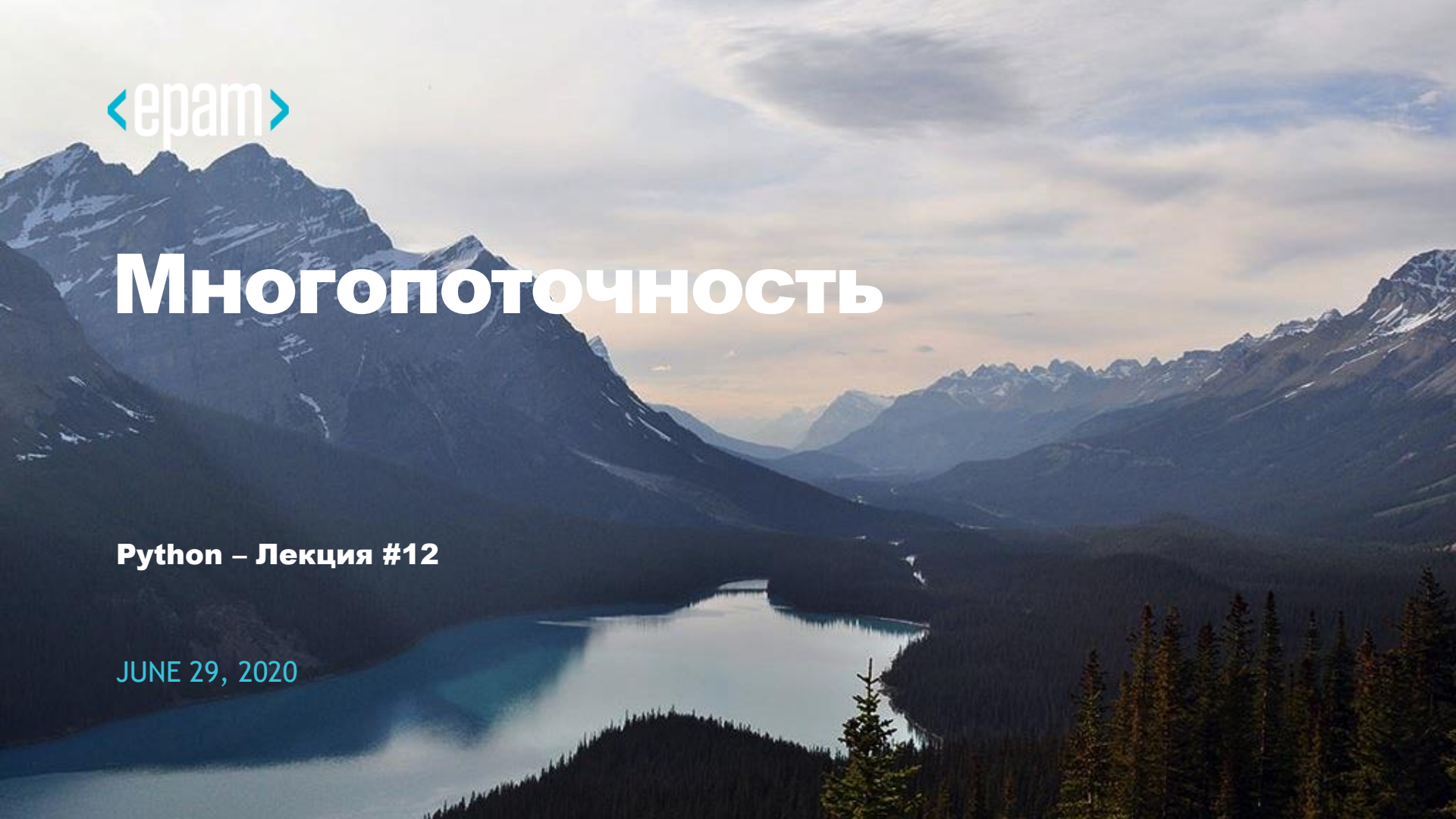


Многопоточность

Python – Лекция #12

JUNE 29, 2020

ТЕМЫ

1. Основные понятия
2. Проблемы синхронизации
3. Глобальная блокировка интерпретатора (GIL)
4. Дополнительные инструменты

Часть 1

ОСНОВНЫЕ ПОНЯТИЯ

ОСНОВНЫЕ ПОНЯТИЯ

Поток (англ. *thread* — нить) — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

В большинстве случаев потоки объединяются в **процессы** (англ. *process*)

На одном процессоре многопоточность обычно реализуется путём временного мультиплексирования (поочередного переключения между разными потоками выполнения, т.н. **переключение контекста**).

При переключении контекста происходит сохранение и восстановление информации о состоянии (например значения регистров процессора), необходимой для продолжения выполнения кода с того же места.

Программный код (компонент ядра ОС), выполняющий назначение приоритетов потокам и процессам, называется **планировщиком** (англ. *scheduler*).

ПРОЦЕСС И ПОТОК

Основные отличия:

- Процессы обычно независимы друг от друга, потоки же существуют как составные части процессов
- Потоки имеют общее адресное пространство, процессы – нет
- Передача данных между процессами сложнее и требует использования специальных средств ОС (межпроцессное взаимодействие)
- Переключение контекста между потоками одного процесса, как правило, происходит быстрее, нежели между потоками разных процессов

НАПОМИНАНИЕ О ВИДАХ МНОГОЗАДАЧНОСТИ

ВЫТЕСНЯЮЩАЯ МНОГОЗАДАЧНОСТЬ — **АНГЛ. PREEMPTIVE MULTITASKING**

- Выполнение каждой задачи происходит в отдельном процессе или потоке
- Передача управления между задачами производится принудительно средствами ОС (переключение контекста)

КООПЕРАТИВНАЯ МНОГОЗАДАЧНОСТЬ — **АНГЛ. COOPERATIVE MULTITASKING**

- Задачи выполняются в одном и том же потоке
- Переключение происходит когда текущая задача явно объявит себя готовой передать управление (англ. *yield control*) другим задачам

ПРИМЕНЕНИЯ МНОГОПОТОЧНОСТИ

Некоторые из применений:

- Задействование всех ядер процессора
- Неблокирующие долгие фоновые операции (например в GUI)
- Выполнение задач в режиме ожидания ввода-вывода

ВЫПОЛНЕНИЕ ЗАДАЧ В РЕЖИМЕ ОЖИДАНИЯ I/O

Задача: написать скрипт, выполняющий действия до момента нажатия клавиши ‘q’

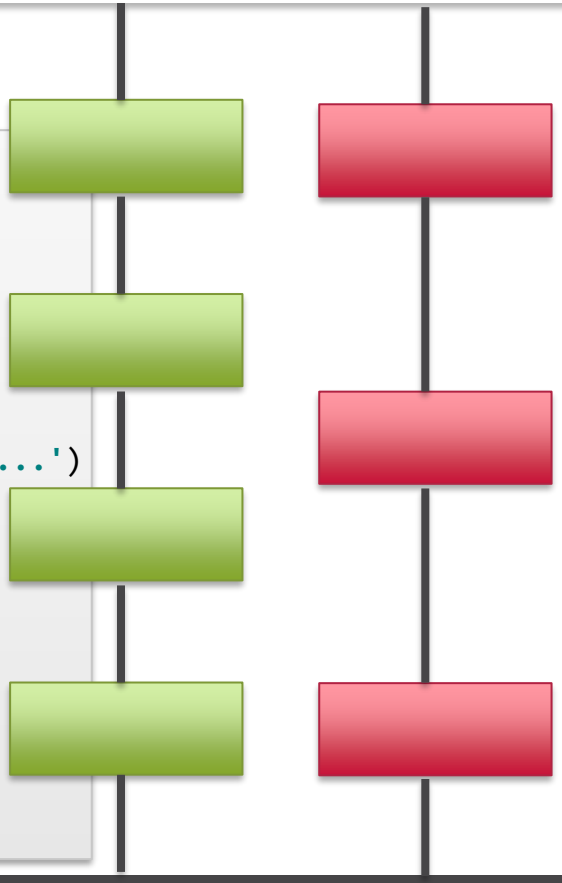
```
import keyboard; import threading; import time

def do_some_operations():
    while True:
        print('Performing operations… Quit: `{}`'.format('q'))
        time.sleep(3)

def graceful_teardown():
    print('Performing teardown operations and stopping execution...')

if __name__ == '__main__':
    thread = threading.Thread(target=do_some_operations)
    thread.daemon = True
    thread.start()

    while True:
        if keyboard.is_pressed('q'):
            graceful_teardown()
            break
```



ПЕРЕДАЧА ПРИВЕТОВ С ПОМОЩЬЮ ПОТОКОВ

Задача: передавать привет Кириллу один раз в две секунды, а Артёму - раз в три секунды.

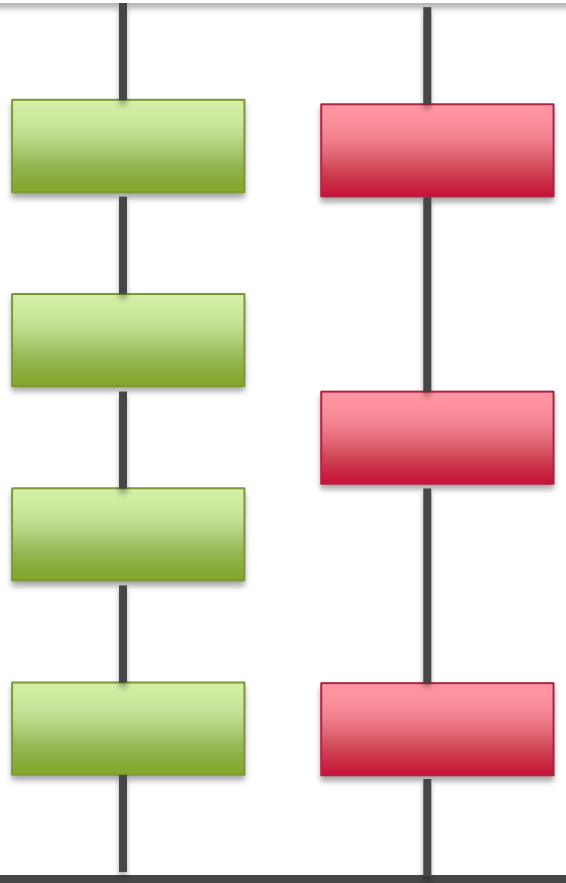
```
from threading import Thread
import time

def hello(name, interval):
    while True:
        print("Hello, %s" % name)
        time.sleep(interval)

t1 = Thread(target=hello, args=("Kirill", 2))
t2 = Thread(target=hello, args=("Artem", 3))

t1.start(); t2.start()
t1.join(); t2.join()
```

Того же самого можно добиться создав дочерний класс от `Thread` и переопределить метод `.run()`



Часть 2

ПРОБЛЕМЫ СИНХРОНИЗАЦИИ

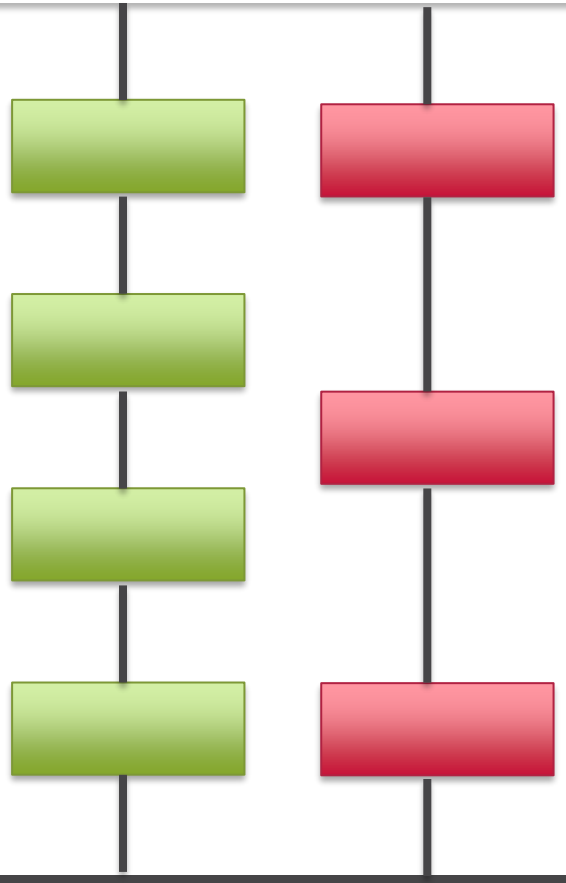
ДОСТУП К ГЛОБАЛЬНОЙ ПЕРЕМЕННОЙ

Увеличение глобальной переменной таким образом уязвимо к ошибке, называемой «race condition», т.к. операция инкремента в данном случае не *атомарна*.

```
import time

counter = 0

def hello(name, interval):
    while True:
        global counter
        counter += 1
        print("Hello, %s (%d)" % (name, counter))
        time.sleep(interval)
```

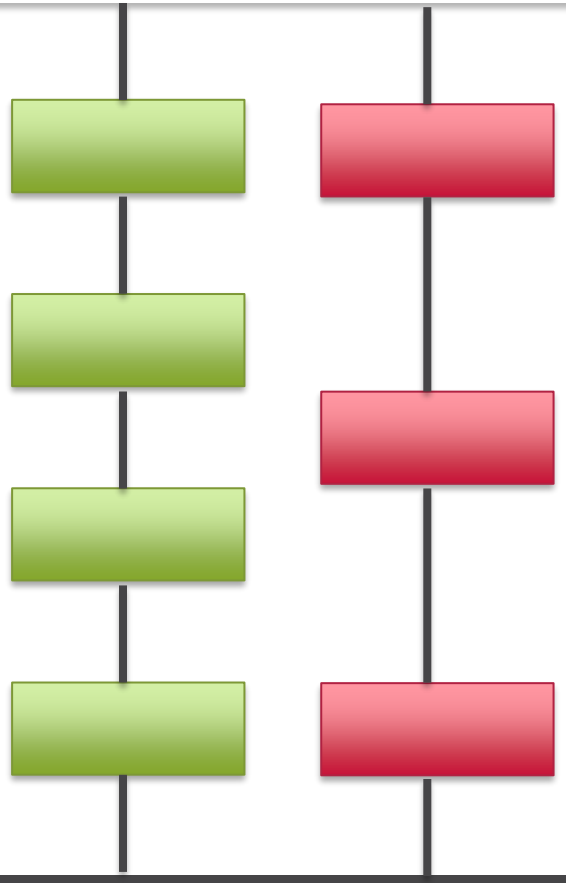


ИНКРЕМЕНТ В БАЙТ-КОДЕ

```
import dis

def incr():
    global x
    x += 1


>>> dis.dis(incr)
 5           0 LOAD_GLOBAL              0 (x)
             2 LOAD_CONST               1 (1)
             4 INPLACE_ADD
             6 STORE_GLOBAL             0 (x)
             8 LOAD_CONST               0 (None)
            10 RETURN_VALUE
```



СОСТОЯНИЕ ГОНКИ ПРИ ИНКРЕМЕНТЕ

Поток 1	Поток 2		Значение *x*
			0
LOAD_GLOBAL (x)		←	0
…	LOAD_GLOBAL (x)	←	0
INPLACE_ADD	…		0
…	INPLACE_ADD		0
STORE_GLOBAL (x)	…	→	1
	STORE_GLOBAL (x)	→	1

СОСТОЯНИЕ ГОНКИ

Состояние гонки (англ. *race condition*) — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

Зачастую состояние гонки — «плавающая» ошибка (гейзенбаг), проявляющаяся в случайные моменты времени и «пропадающая» при попытке её локализовать.

ИСПОЛЬЗОВАНИЕ БЛОКИРОВКИ

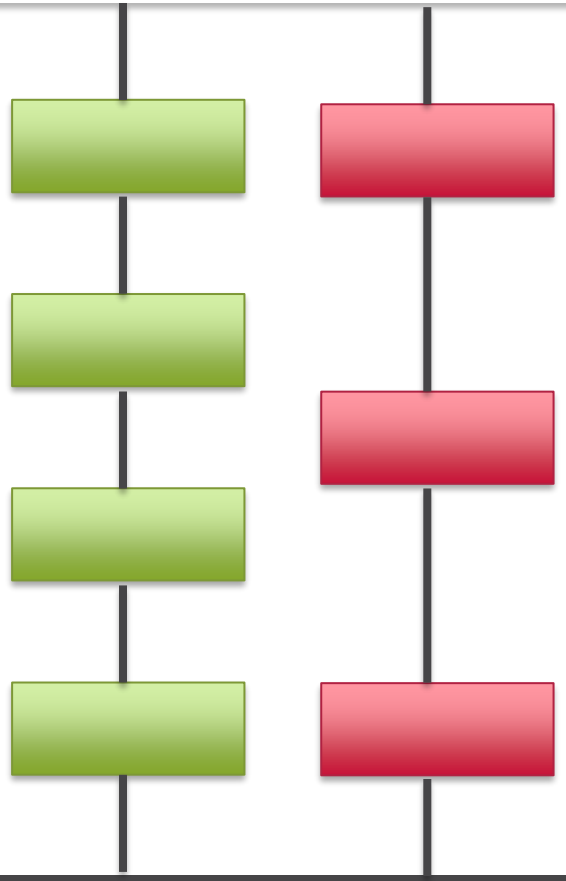
Версия функции **hello** без “race condition”.

```
from threading import Lock
import time

counter = 0; lock = Lock()

def hello(name, interval):
    global counter
    while True:
        lock.acquire()
        counter += 1
        lock.release()
        print("Hello, %s (%d)" % (name, counter))
        time.sleep(interval)
```

Объект `Lock()` также можно использовать как контекстный менеджер вместо ручного вызова `acquire()` и `release()`. Это рекомендуемый способ.



ПРОБЛЕМА - ВЗАИМНАЯ БЛОКИРОВКА

Взаимная блокировка (англ. *deadlock*) — ситуация в многозадачной среде, при которой несколько процессов или потоков находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать свое выполнение.

Шаг	Поток 1	Поток 2
0	Хочет захватить A и B, начинает с A	Хочет захватить A и B, начинает с B
1	Захватывает ресурс A	Захватывает ресурс B
2	Ожидает освобождения ресурса B	Ожидает освобождения ресурса A
3	Взаимная блокировка	

Такая проблема решается усложнением механизма блокировок (например введении их «иерархии»).

Часть 3

ГЛОБАЛЬНАЯ БЛОКИРОВКА ИНТЕРПРЕТАТОРА

СОСТОЯНИЕ ГОНКИ В СЧЕТЧИКЕ ССЫЛОК

Поток 1	Поток 2		Значение obj_refcount
			0
MOV obj_refcount, AX		←	0
INC AX			0
MOV AX, obj_refcount		→	1
	MOV obj_refcount, AX	←	1
	INC AX		1
	MOV AX, obj_refcount	→	2

СОСТОЯНИЕ ГОНКИ В СЧЕТЧИКЕ ССЫЛОК

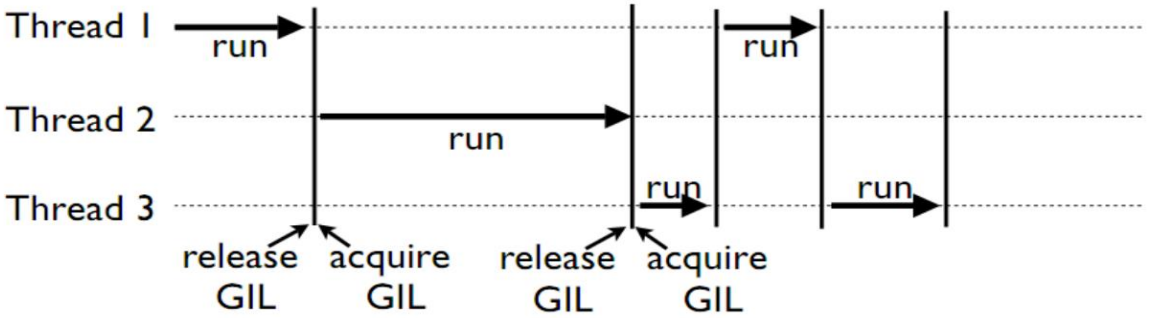
Поток 1	Поток 2		Значение *obj_refcount*
			0
MOV *obj_refcount*, AX		←	0
	MOV *obj_refcount*, AX	←	0
INC AX			0
	INC AX		0
MOV *obj_refcount*, AX		→	1
	MOV AX, *obj_refcount*	→	1

Возможные последствия:

- утечка памяти (англ. *memory leak*)
- завершение программы с ошибкой доступа к памяти (*Segmentation Fault* в Linux)
- некорректное поведение программы

GLOBAL INTERPRETER LOCK

Глобальная блокировка интерпретатора (англ. *Global interpreter lock – GIL*) — окончательное решение проблемы потокобезопасности интерпретатора. Реализуется средствами ОС (мьютексом).



GIL присутствует не во всех реализациях Python (например в Jython и IronPython его нет)

«CPU-BOUND CODE» И «I/O-BOUND CODE»

“CPU-bound code” – код, который преимущественно не выходит за пределы интерпретатора.

“I/O-bound code” – код, который содержит большое количество блокирующих операций ввода вывода. Перед такими операциями интерпретатор «освобождает» GIL, давай другим потокам возможность выполниться.

GIL существенно ограничивает параллелизм для “CPU-bound code”, и оказывает мало влияния на параллелизм в случае “I/O-bound code”.

GIL И РАСШИРЕНИЯ НА ЯЗЫКЕ C

```
Py_BEGIN_ALLOW_THREADS          // «release the GIL» (макрос)

// какие либо вычисления, не связанные со взаимодействием с интерпретатором

// например, сон
sleep(10)

// или чтение или запись в файл
write(fd, buffer, size)

// или чтение или запись в сокет
recv(sock, buffer, size, flags)

Py_END_ALLOW_THREADS            // «acquire the GIL» (макрос)
```

ПРОБЛЕМЫ GIL

Ограничения параллелизма – это не проблема GIL, это особенность реализации, которую нужно принимать во внимание при решении задач.

Самые большие проблемы GIL кроются в его взаимодействии с планировщиком ОС.

- Планировщик ОС отдает приоритет для “CPU-bound threads”
- На многоядерных процессорах CPU-bound потоки осуществляют ожесточенную борьбу за GIL между собой, что приводит к падению производительности в разы (последствия такой борьбы немного смягчены в «новом» GIL в Python 3.2, но в целом проблема остается)

ПОЧЕМУ GIL ВСЕ ЕЩЕ ЕСТЬ В CPYTHON?

Основные причины:

- такое устройство интерпретатора оптимально для однопоточных программ
- облегчается интеграция стороннего кода на C, который зачастую непотокобезопасен

> “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.”
>
> “It isn’t Easy to Remove the GIL”,
> Guido van Rossum, September 10, 2007

Часть 4

ДОПОЛНИТЕЛЬНЫЕ ИНСТРУМЕНТЫ

ОБЗОР ДОПОЛНИТЕЛЬНЫХ ИНСТРУМЕНТОВ

- Мьютексы
- «Reentrant locks»
- Семафоры
- События
- Очереди
- Пулы потоков

МЬЮТЕКСЫ

Мьютекс (англ. *mutex*, от *mutual exclusion* — «взаимное исключение») — простейший механизм блокировки общих данных от одновременного доступа.

В Python мьютексы реализуются с помощью ранее рассмотренного нами класса `Lock`

```
import threading

lock = threading.Lock()
lock.acquire()        # захват

# работа с общими данными
do_something()


lock.acquire()        # освобождение
```

REENTRANT LOCKS

Класс *RLock* – это версия замка, который выполняет функцию блокировки только в том случае, если замок удерживает другой поток.

В то время как обычные замки блокируют тогда, когда тот же поток пытается получить к одному и тому же замку дважды, реентерабельный замок блокирует только в том случае, если другой поток уже держит замок. Если нынешний поток пытается получить доступ к замку, который и так удерживается, осуществление данной операции проходит в привычном порядке.

```
import threading

lock = threading.Lock()
lock.acquire()
lock.acquire()      # заблокирует

lock = threading.RLock()
lock.acquire()
lock.acquire()      # не будет блокировать
```

СЕМАФОРЫ

Семафор (англ. *semaphore*) — объект, ограничивающий количество потоков, которые могут войти в заданный участок кода.

К примеру, семафор может быть использован для доступа к некому подключению с ограниченной пропускной способностью.

`BoundedSemaphore` считает ошибкой вызов `release()` больше раз, чем был вызван `acquire()`

```
import threading

semaphore = threading.BoundedSemaphore(10)
semaphore.acquire()       # уменьшает счетчик (-1)

# доступ к общим ресурсам
semaphore.release()       # увеличивает счетчик (+1)
```

СОБЫТИЯ

Объект `Event()` – это простой объект синхронизации. Он представляет собой внутренний флаг, так что все потоки могут ожидать, пока флаг будет установлен, задавать, или убирать его.

```
import threading

event = threading.Event()

# поток ожидает установки флага
event.wait()

# другой поток может установить флаг или снять его.
event.set()
event.clear()
```

ОЧЕРЕДИ

Модуль queue содержит различные реализации очередей, которые удобно использовать для организации безопасной передачи данных между потоками.

В примере ниже используется FIFO-очередь (но есть также и LIFO)

```
import queue

q = queue.Queue()

# Помещаем элементы в очередь
for x in range(4):
    q.put("Item %s" % x)

# Извлекаем элементы из очереди
while not q.empty():
    print(q.get())
```

ПУЛЫ ПОТОКОВ

Пул потоков (англ. Thread pool) – инструмент, позволяющий иметь открытыми фиксированное количество потоков и обрабатывать большой массив данных с их помощью.

В такой варианте использование вычислительных ресурсов становится более предсказуемым. Также не расходуются ресурсы на пересоздание отдельного потока для каждого обрабатываемого элемента.

```
from multiprocessing.pool import ThreadPool

def print_quote(quote):
    print(quote.upper())

quotes = [
   'Quos Deus vult perdere, prius dementat',
   'Another quote here...',
   'Yet another ancient wisdom...',
]

# Количество потоков всегда постоянно постоянно
pool = ThreadPool(2)

# Обрабатывать каждый элемент в отдельном потоке
results = pool.map(print_quote, quotes)

# Дождаться обработки всех значений
pool.close(); pool.join()
```

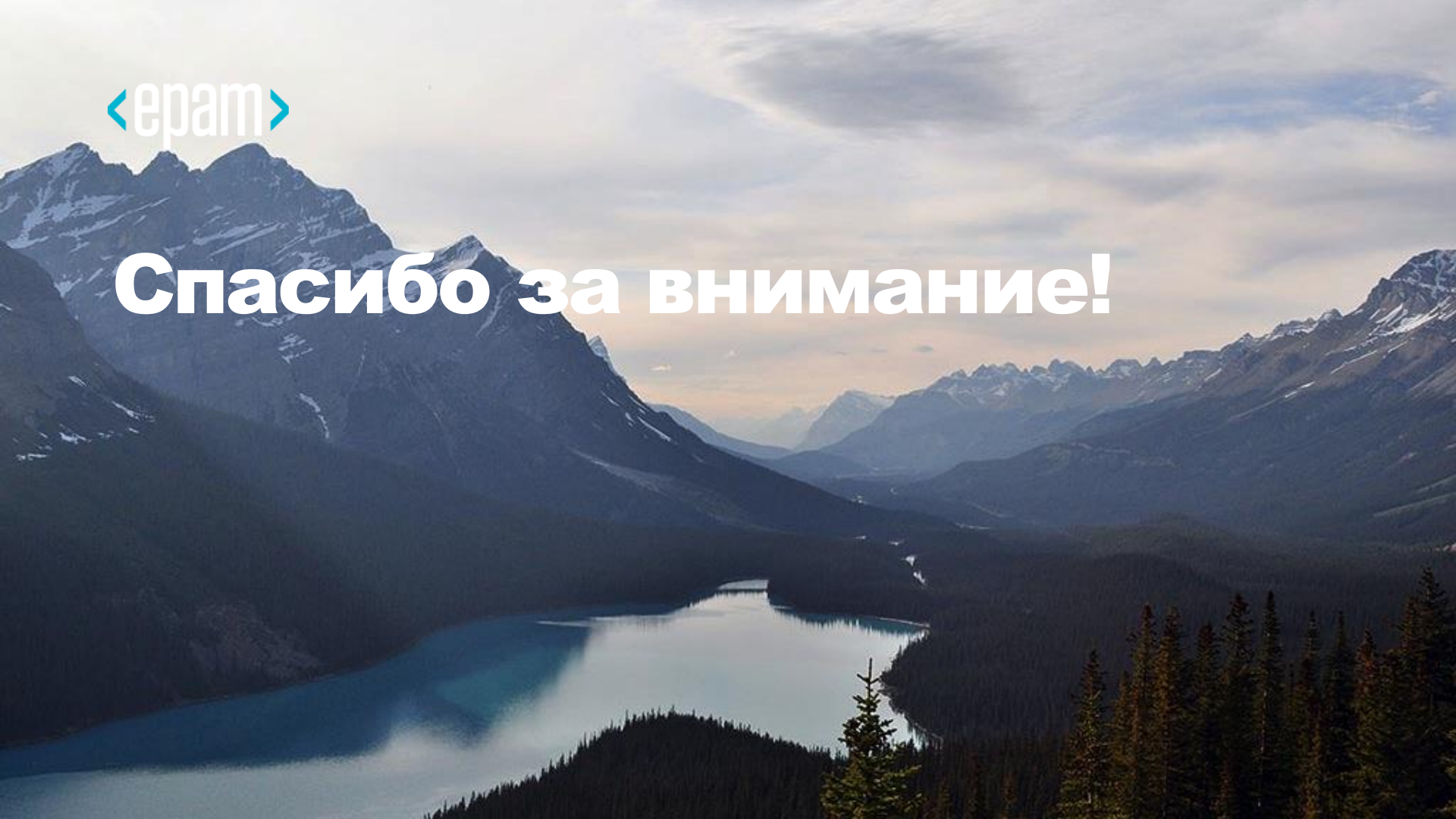
ДОМАШНЕЕ ЗАДАНИЕ

Написать консольную утилиту для многопоточного скачивания файлов, их последующей обработки и сохранения в локальной файловой системе. Обработка файлов заключается в генерации превью заданного максимального размера (с сохранением соотношения сторон) и перекодировании в формат JPEG.

Требования
- Файл со списком URL указывается параметром в командной строке (по одному URL в каждой строке). Если файл не существует, выводится сообщение об ошибке.
- Количество потоков, размеры выходных картинок и выходной каталог задаются параметрами в командной строке и имеют значения по умолчанию (один поток, запись в текущий каталог, размеры 100x100). Если параметры будут зашиты в скрипт, то будут сняты баллы.
- Если выходной каталог не существует, его нужно автоматически создать.
- Файлы сохраняются под именами, соответсвующими их номеру строки в списке URL (00000.jpeg, 00001.jpeg и т.д.). Допустимо перезаписывать файлы существующие в каталоге на момент запуска без предупреждения. Конкатенация путей должна быть платформо-независимой (на Windows и Linux разделителями в путях файлов служат разные слеши, и для безопасного построения путей нужно использовать os.path.join)
- Промежуточные (непережатые) файлы всегда держатся в памяти, не нужно сохранять их на диск.
- Невалидные URL, ошибки скачивания файлов и файлы нечитаемых форматов должны корректно обрабатываться (сами файлы игнорировать, только увеличивать счетчик ошибок).
- В ходе работы в стандартный вывод выводятся сообщения о завершении обработки каждого файла. После завершения в стандартный вывод выводится статистика - количество скачанных файлов, количество скачанных байт, количество запросов завершившихся с ошибкой, общее время выполнения.
- Должен присутствовать файл requirements.txt со списком зависимостей для установки их через pip
- Unit-тесты в этом задании писать не нужно

Пример использования: скачать все файлы из списка urllist.txt в 4 потока, сохранить в каталог thumbnails в размере 128x128 пикселей.
python download.py urllist.txt --dir=thumbnails/ --threads=4 --size=128x128

Подсказки
Для парсинга аргументов командной строки можно использовать модуль argparse из стандартной библиотеки https://docs.python.org/3/library/argparse.html.
Для скачивания файлов можно использовать встроенную библиотеку urllib https://docs.python.org/3/library/urllib.html
Для обработки изображений можно использовать стороннюю библиотеку Pillow https://pillow.readthedocs.io/en/5.1.x/
Для управления потоками можно использовать ThreadPool из стандартной библиотеки, для передачи данных между потоками - Queue. Не забывайте про потокобезопасность.


epam
Спасибо за внимание!