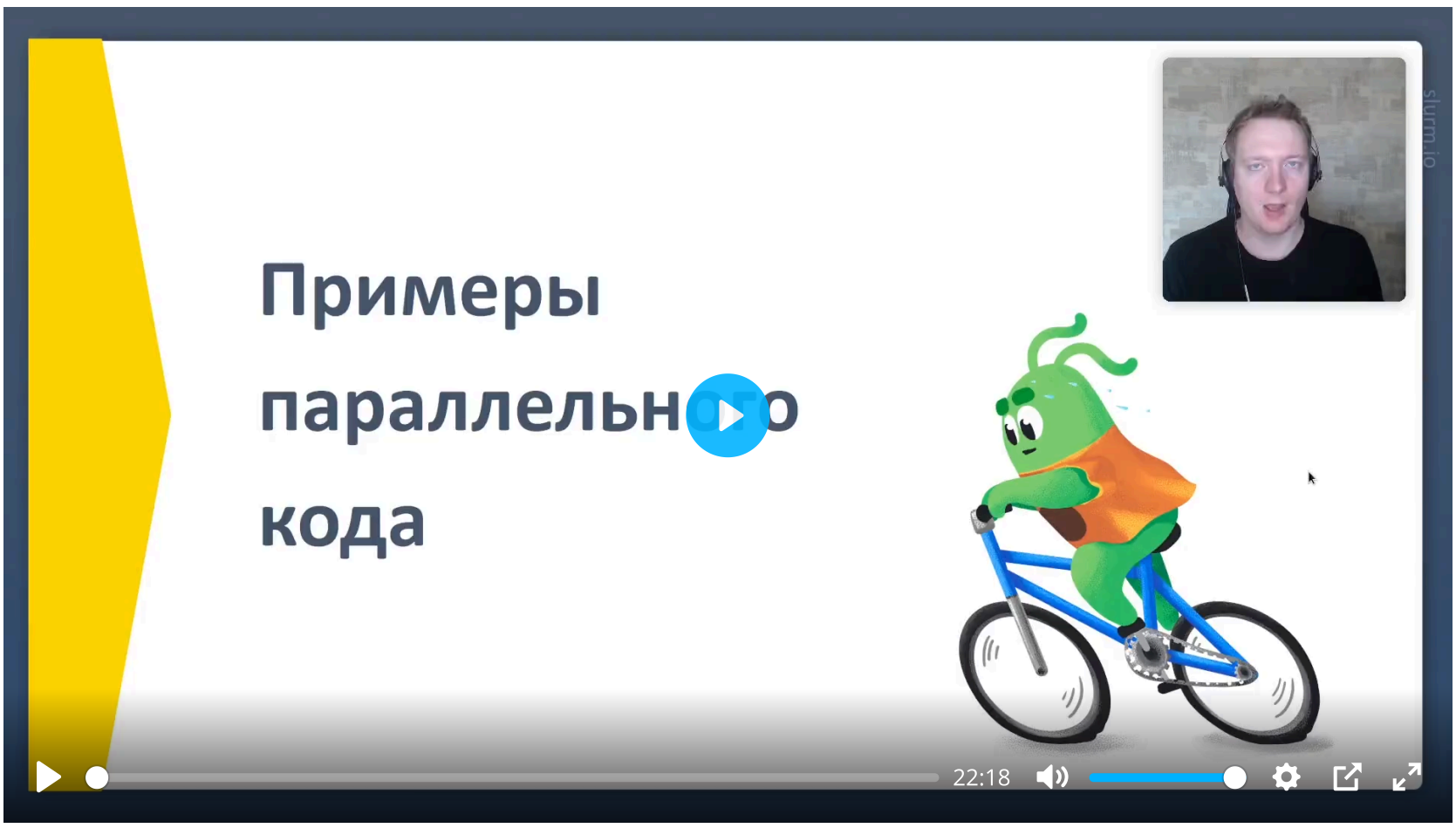
2.3 Параллельность и асинхронность в Python



Текстовая расшифровка видео:

ПРИМЕРЫ ПАРАЛЛЕЛЬНОГО КОДА

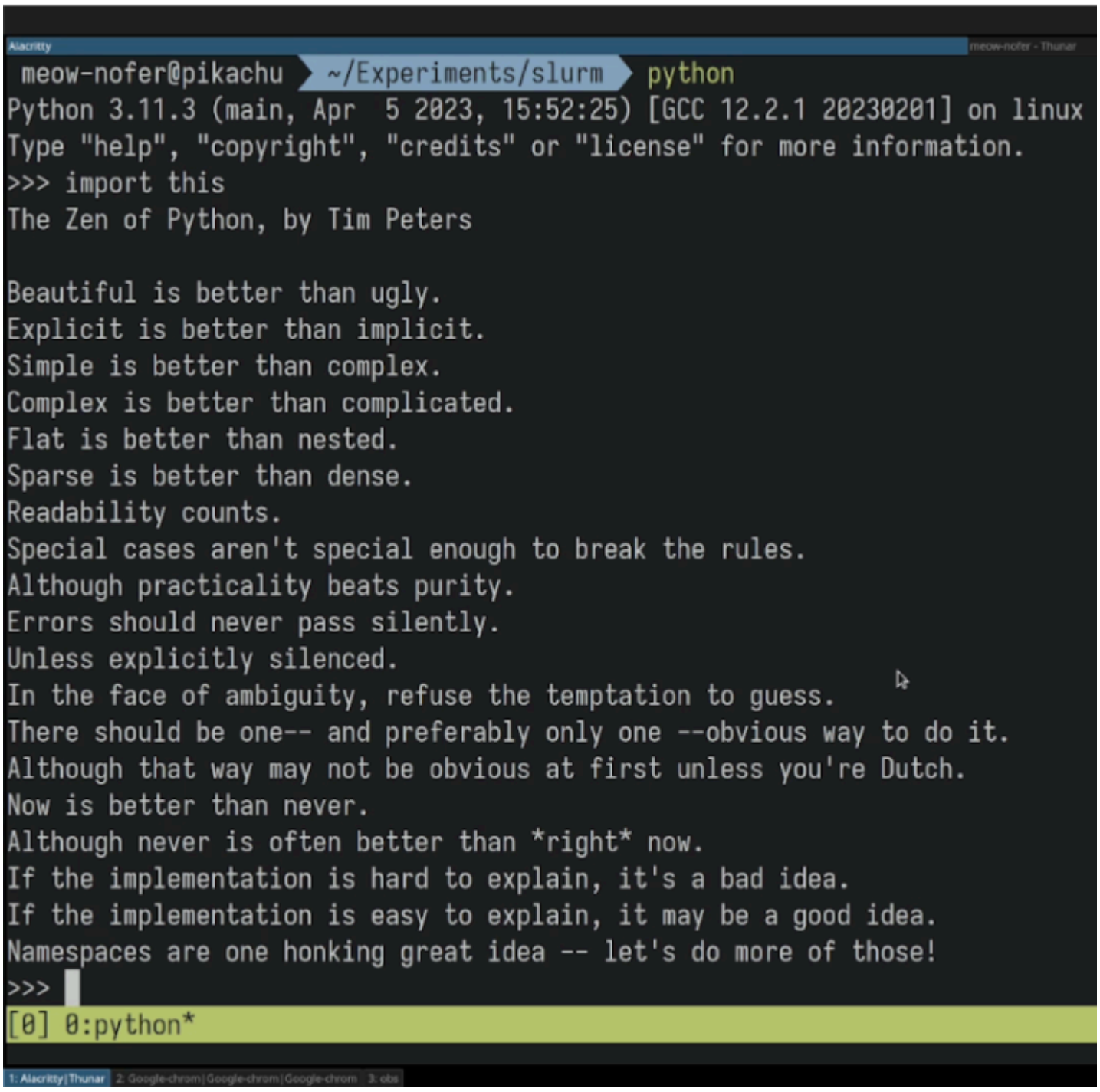
План:

- Примеры многопоточного кода;
- А можно просто взять процессы?

Примеры многопоточного кода

Нам известно, что в питоне есть понятие «Дзен Python'а».

Если мы запустим интерпретатор и напишем «import this», то Python напишет нам «Дзен-набор» правил, которым, с точки зрения разработчиков, стоит следовать:



Один из пунктов, который здесь есть гласит: «В идеале должен быть один и только один способ в решении какой-либо задачи».

Ирония заключается в том, что с развитием Python многое стало конфликтовать с предложенными постулатами, в частности при работе с многопоточностью. Для работы с потоками в Python есть два взаимозаменяемых механизма – модули **threading** и **concurrent.futures**.

Рассмотрим на примерах:

```python
import os
import queue

import requests

from threading import Thread


class DownloadThread(Thread):

    def __init__(self, queue, name):
        super().__init__()
        self.queue = queue
        self.name = name

    def run(self):
        while True:
            url = self.queue.get()
            fname = os.path.basename(url)

            res = requests.get(url, stream=True)
            res.raise_for_status()
```

```
NORMAL  threading_example.py                                    python
"threading_example.py" 51L, 1276B
[0] 0:nvim*
```

Начнем с кода на потоках. Когда мы хотим запускать потоки, мы хотим что-либо реализовывать в параллели. Так, нам нужно создать объекты потока и каждому из них задать функцию, которую он будет выполнять, с точки зрения API, который нам представляет reading thread. От него наследуемся. Далее, в классе для нашего объекта реализуем метод «Run». Данный метод включает в себя функцию, которая будет выполняться в потоке и может выполняться в параллели с другими потоками.

Что происходит с кодом: у нас есть стандартная очередь (пакет queue), из нее мы достаем адреса (url) и с помощью библиотеки requests выкачиваем файлы, находящиеся по этим адресам, а потом сохраняем их на диск. Далее, рапортуем в очередь о выполнении задачи:

```
Alacritty                                                    meow-nofer - Thunar
threading_example.py
10
11      def __init__(self, queue, name):
12          super().__init__()
13          self.queue = queue
14          self.name = name
15
16      def run(self):
17          while True:
18              url = self.queue.get()
19              fname = os.path.basename(url)
20
21              res = requests.get(url, stream=True)
22              res.raise_for_status()
23
24              with open(fname, "wb") as savefile:
25                  for chunk in res.iter_content(1024):
26                      savefile.write(chunk)
27
28              self.queue.task_done()
29              print(f"{self.name} finished downloading {url} !")
30
31
32  def main(urls):
NORMAL  threading_example.py                                  python
[0] 0:nvim*
1: Alacritty | Thunar  2: Google-chrom | Google-chrom | Google-chrom  3: obs
```

Это код, который будет запущен в дочернем потоке (не основном). В основном потоке мы сделаем экземпляр нашей очереди и создадим пачку потоков. В качестве примера создадим два потока. Пройдясь по ним, запускаем. Далее, в нашу очередь записываем url и дожидаемся окончания task done.

Частая ошибка новичков в работе с многопоточностью: мы запускаем потоки с помощью метода «.start», а код реализуем с помощью метода **«run»**.

Не путайте эти методы: не дергайте «run» и не реализовывайте «.start»!

На примере вы можете видеть формы в качестве файлов, взятых с сайта американской налоговой службы:

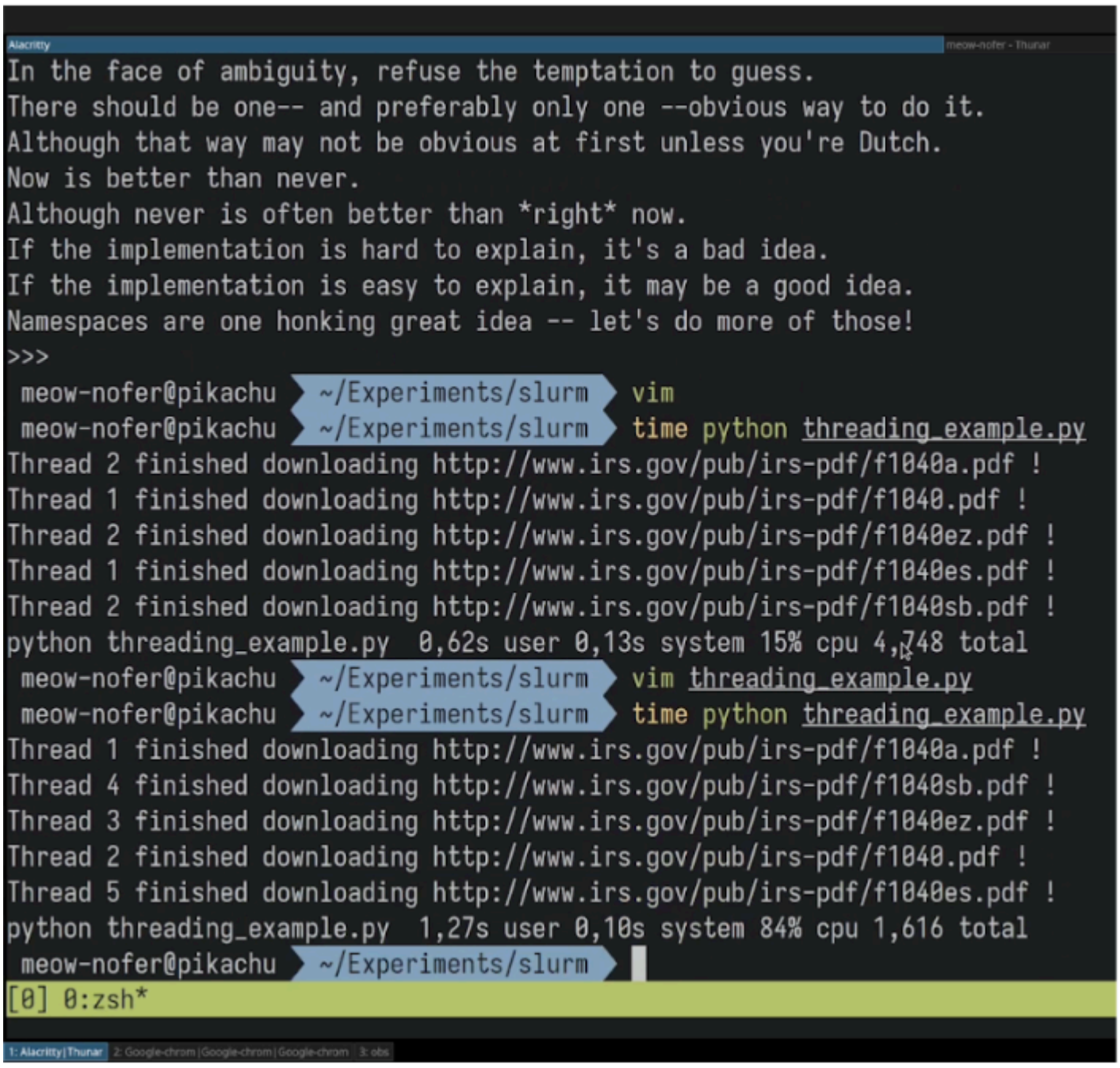
```python
def main(urls):
    q = queue.Queue()
    threads = [DownloadThread(q, f"Thread {i + 1}") for i in range(2)]
    for t in threads:
        # not waiting for child threads
        t.daemon = True
        t.start()

    for url in urls:
        q.put(url)

    q.join()  # all cheeki-breeki

main([
    "http://www.irs.gov/pub/irs-pdf/f1040.pdf",
    "http://www.irs.gov/pub/irs-pdf/f1040a.pdf",
    "http://www.irs.gov/pub/irs-pdf/f1040ez.pdf",
    "http://www.irs.gov/pub/irs-pdf/f1040es.pdf",
    "http://www.irs.gov/pub/irs-pdf/f1040sb.pdf"
])
```

Мы можем, используя команду «time», замерить сколько затрачивается времени на скачивание этих файлов (помним о том, что у нас два потока).

После рапорта потоков мы видим результат в виде 4,7 секунд. Данный результат выявлен на двух потоках:

```
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
 meow-nofer@pikachu  ~/Experiments/slurm  vim
 meow-nofer@pikachu  ~/Experiments/slurm  time python threading_example.py
Thread 2 finished downloading http://www.irs.gov/pub/irs-pdf/f1040a.pdf !
Thread 1 finished downloading http://www.irs.gov/pub/irs-pdf/f1040.pdf !
Thread 2 finished downloading http://www.irs.gov/pub/irs-pdf/f1040ez.pdf !
Thread 1 finished downloading http://www.irs.gov/pub/irs-pdf/f1040es.pdf !
Thread 2 finished downloading http://www.irs.gov/pub/irs-pdf/f1040sb.pdf !
python threading_example.py  0,62s user 0,13s system 15% cpu 4,748 total
 meow-nofer@pikachu  ~/Experiments/slurm
[0] 0:zsh*
```

Попробуем увеличить количество потоков. Вспомним, что в Python при выполнении byte-code нет настоящей параллельности, наша задача – скачать файлы. Это задача I/O bound, так как мы ходим через сетевую карточку и запрашиваем у системы возможность читать из сетевого сокета и записывать данные на диск. У нас должен получится прирост производительности при использовании большого количества потоков. Например, если мы возьмем пять потоков и запустим их, то у нас все выполнится практически моментально:



Для чистоты эксперимента снесем файлы и убедимся в скорости работы:

```
Alacritty                                                    meow-nofer - Thunar
>>>
 meow-nofer@pikachu  ~/Experiments/slurm  vim
 meow-nofer@pikachu  ~/Experiments/slurm  time python threading_example.py
Thread 2 finished downloading http://www.irs.gov/pub/irs-pdf/f1040a.pdf !
Thread 1 finished downloading http://www.irs.gov/pub/irs-pdf/f1040.pdf !
Thread 2 finished downloading http://www.irs.gov/pub/irs-pdf/f1040ez.pdf !
Thread 1 finished downloading http://www.irs.gov/pub/irs-pdf/f1040es.pdf !
Thread 2 finished downloading http://www.irs.gov/pub/irs-pdf/f1040sb.pdf !
python threading_example.py  0,62s user 0,13s system 15% cpu 4,748 total
 meow-nofer@pikachu  ~/Experiments/slurm  vim threading_example.py
 meow-nofer@pikachu  ~/Experiments/slurm  time python threading_example.py
Thread 1 finished downloading http://www.irs.gov/pub/irs-pdf/f1040a.pdf !
Thread 4 finished downloading http://www.irs.gov/pub/irs-pdf/f1040sb.pdf !
Thread 3 finished downloading http://www.irs.gov/pub/irs-pdf/f1040ez.pdf !
Thread 2 finished downloading http://www.irs.gov/pub/irs-pdf/f1040.pdf !
Thread 5 finished downloading http://www.irs.gov/pub/irs-pdf/f1040es.pdf !
python threading_example.py  1,27s user 0,10s system 84% cpu 1,616 total
 meow-nofer@pikachu  ~/Experiments/slurm  rm *.pdf
 meow-nofer@pikachu  ~/Experiments/slurm  time python threading_example.py
Thread 4 finished downloading http://www.irs.gov/pub/irs-pdf/f1040sb.pdf !
Thread 1 finished downloading http://www.irs.gov/pub/irs-pdf/f1040ez.pdf !
Thread 2 finished downloading http://www.irs.gov/pub/irs-pdf/f1040a.pdf !
Thread 3 finished downloading http://www.irs.gov/pub/irs-pdf/f1040.pdf !
Thread 5 finished downloading http://www.irs.gov/pub/irs-pdf/f1040es.pdf !
python threading_example.py  1,31s user 0,09s system 86% cpu 1,617 total
 meow-nofer@pikachu  ~/Experiments/slurm
[0] 0:zsh*
1: Alacritty | Thunar  2: Google-chrom | Google-chrom | Google-chrom  3: obs
```

Вывод: **поток** – системное понятие, в котором системный шедулер переключает контексты выполнений. Не создавайте больше потоков, чем ваше количество ядер процессоров. Система не сможет физически выполнять данный запрос. Например, если у вас 16 ядер, то не стоит создавать более 16 потоков.

Рассмотрим данный параметр:

```
37                t.daemon = True
```

В приложении есть большое количество потоков: основной поток и дочерние. Вопрос: что делать, когда основной поток завершился, а дочерние еще нет?

Есть два варианта:

1) Дожидаемся в основном потоке завершения дочерних;

2) Не дожидаемся завершения дочерних потоков, проверяя обработанность всех записей. Когда завершился join в основном потоке, мы знаем, что полезной работы от дочерних потоков уже не будет, поэтому мы их пробиваем. Параметр «daemon» отвечает именно за это. Мы помечаем потоки как те, которые можно пробить.

Пример использования библиотеки на concurrent.futures:

```python
import concurrent.futures
import os

import requests

def load_url(url):
    fname = os.path.basename(url)
    res = requests.get(url, stream=True)
    res.raise_for_status()

    with open(fname, "wb") as savefile:
        for chunk in res.iter_content(1024):
            savefile.write(chunk)
    return fname

URLS = [
    "http://www.irs.gov/pub/irs-pdf/f1040.pdf",
    "http://www.irs.gov/pub/irs-pdf/f1040a.pdf",
    "http://www.irs.gov/pub/irs-pdf/f1040ez.pdf",
    "http://www.irs.gov/pub/irs-pdf/f1040es.pdf",
    "http://www.irs.gov/pub/irs-pdf/f1040sb.pdf"
]
```

```
NORMAL  futures_example.py                                    python
"futures_example.py" 31L, 929B
[0] 0:nvim*
```

Идея та же самая: необходимо создать потоки и каждому из них указать функцию. Однако здесь мы будем использовать механизм, который похож на механизм, реализующий thread pool.

Функции по скачиванию файлов идентичны и отличаются лишь тем, что здесь нет бесконечного цикла:

```python
def load_url(url):
    fname = os.path.basename(url)
    res = requests.get(url, stream=True)
    res.raise_for_status()

    with open(fname, "wb") as savefile:
        for chunk in res.iter_content(1024):
            savefile.write(chunk)
    return fname
```

Что делаем: мы создаем при помощи concurrent.futures thread pool. Это делается при помощи обертки «TreadPoolExecutor». В этом pool’е мы создаем максимальное количество фиксированных потоков, после чего начинаем закидывать задачи при помощи метода «submit». Submit мы передаем функцию и ее аргументы:

```python
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    future_to_url = {
        executor.submit(load_url, url): url
        for url in URLS
    }
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        print(f"URL '{future_to_url[future]}' is saved to '{future.result()}'")
```

Хитрость: использовать submit в качестве ключа в словаре, где значение в словаре – url, который нужно скачать.

Когда мы проходимся и сабмитим задачу в thread pool, из сабмита возвращается специальный тип объекта, который называется «future».

Future – это объект, в который в будущем придет результат.

Далее мы смотрим на список ключей и используем обертку «as_completed». Мы ждем возвращения результата хотя бы одним из фьючеров. Как только это происходит, у нас проматывается итерация цикла. Из исходного маппинга мы достаем url и выводим.

А можно просто взять процессы?

Вопрос: а можно просто взять процессы?

Ответ: можно. Поскольку у нас есть GIL, нам придется использовать процессы, а не потоки. Мы можем использовать встроенный механизм операционной системы, когда основной процесс порождает дочерний.

У Python есть собственные встроенные модули для работы с процессами (схожи с модулями для работы с потоками):

- **Multiprocessing**, где объектом является «Process» и реализация очереди «Queue»;
- **ProcessPoolExecutor**.

В коде нам нужно будет заменить «Tread» на «Process» и очередь на «Multiprocessing» и «Queue», а также «TreadPoolExecutor» на «ProcessPoolExecutor».

Нюанс: в случае использования процессов возникают большие затраты на пересылку данных и сериализацию. Над этим уже ведется работа.

Есть механизм, который позволяет (в частности, в Linux) более эффективно передавать данные из процесса в процесс. Данный механизм называется «**Shared memory**». Это возможность, позволяющая создать в оперативной памяти общий кусочек для нескольких процессов.

В Python для того, чтобы это работало, в стандартной библиотеке есть модуль «multiprocessing.shared_memory». В текущем виде этот модуль слишком простой и позволяет пересылать данные только простых типов. Для серьезных продуктовых решений на сегодняшний день его использовать затруднительно.

Ознакомиться можно здесь.