Процесс и его характеристики

- Что такое процесс?
- Какие процессы запущены в ОС?
- Как запустить python процесс?
- Что делает процесс во время исполнения?

Характеристики процесса:

- Идентификатор процесса, PID
- Объем оперативной памяти
- Стек
- Список открытых файлов
- Ввод/вывод

In [ ]:
```
# простой Python процесс

import time
import os

pid = os.getpid()

while True:
    print(pid, time.time())
    time.sleep(2)

> $ python ex1.py
> 15468 1488521934.518766
> 15468 1488521936.520758
> 15468 1488521938.522762
> ...
```

Создание процесса на Python

- Как создать дочерний процесс?
- Как работает системный вызов fork?
- Модуль multiprocessing

In [ ]:

```
# Создание процесса на Python

import time
import os

pid = os.fork()
if pid == 0:
    # дочерний процесс
    while True:
        print("child:", os.getpid())
        time.sleep(5)
else:
    # родительский процесс
    print("parent:", os.getpid())
    os.wait()

> $ python ex2.py
> parent: 14689
> child: 14690
```

In [ ]:

```
# Память родительского и дочернего процесса

import os

foo = "bar"

if os.fork() == 0:
    # дочерний процесс
    foo = "baz"
    print("child:", foo)
else:
    # родительский процесс
    print("parent:", foo)
    os.wait()

> $ python ex3.py
> parent: bar
> child: baz
```

In [ ]:

```
# Файлы в родительском и дочернем процессе

# $ cat data.txt
# example string1
# example string2

import os

f = open("data.txt")
foo = f.readline()

if os.fork() == 0:
    # дочерний процесс
    foo = f.readline()
    print("child:", foo)
else:
    # родительский процесс
    foo = f.readline()
    print("parent:", foo)

> $ python ex4.py
> parent: example string2
> child: example string2
```

In [ ]:

```
# Создание процесса, модуль multiprocessing

from multiprocessing import Process

def f(name):
    print("hello", name)

p = Process(target=f, args=("Bob",))
p.start()
p.join()

> $ python ex5.py
> hello Bob
```

In [ ]:

```
# Создание процесса, модуль multiprocessing

from multiprocessing import Process

class PrintProcess(Process):
    def __init__(self, name):
        super().__init__()
        self.name = name

    def run(self):
        print("hello", self.name)

p = PrintProcess("Mike")
p.start()
p.join()

> $ python ex6.py
> hello Mike
```

Создание потоков

- Что такое поток
- Создание потоков, модуль threading
- Использование ThreadPoolExecutor

Создание потоков

- Поток напоминает процесс
- У потока своя последовательность инструкций
- Каждый поток имеет собственный стек
- Все потоки выполняются в рамках процесса
- Потоки разделяют память и ресурсы процесса
- Управлением выполнением потоков занимается ОС
- Потоки в Python имеют свои ограничения

In [ ]:

```
# Создание потока

from threading import Thread

def f(name):
    print("hello", name)

th = Thread(target=f, args=("Bob",))
th.start()
th.join()

> $ python ex1.py
> hello Bob
```

In [ ]:

```
# Создание потока

from threading import Thread

class PrintThread(Thread):
    def __init__(self, name):
        super().__init__()
        self.name = name

    def run(self):
        print("hello", self.name)

th = PrintThread("Mike")
th.start()
th.join()

> $ python ex2.py
> hello Mike
```

In [ ]:
```
# Пул потоков, concurrent.futures.Future

from concurrent.futures import ThreadPoolExecutor, as_completed

def f(a):
    return a * a

# .shutdown() in exit
with ThreadPoolExecutor(max_workers=3) as pool:
    results = [pool.submit(f, i) for i in range(10)]

    for future in as_completed(results):
        print(future.result())

$ python ex3.py
0
1
4
9
...
```

Синхронизация потоков

- Очереди
- Блокировки
- Условные переменные

In [ ]:
```
В многопоточной программе доступ к объектам иногда нужно синхронизировать.
Часто для синхронизации потоков используют блокировки.
Любые блокировки замедляют выполнение программы.

Лучше избегать использование блокировок
и отдавать предпочтение обмену данными через очереди.
```

In [ ]:

```
# Очереди, модуль queue
from queue import Queue
from threading import Thread

def worker(q, n):
    while True:
        item = q.get()
        if item is None:
            break
        print("process data:", n, item)

q = Queue(5)
th1 = Thread(target=worker, args=(q, 1))
th2 = Thread(target=worker, args=(q, 2))
th1.start(); th2.start()

for i in range(50):
    q.put(i)

q.put(None); q.put(None)
th1.join(); th2.join()

> $ python ex_queue.py
> process data: 1 0
> process data: 1 1
> process data: 2 3
...
```

In [ ]:

Использование очередей для потоков выглядит как показано на слайде.
Создаем очередь с максимальным размером 5.
Используем методы put() для того чтобы поместить данные в очередь
и get() для того чтобы забрать данные из очереди

Использование очередей делает код выполняемой программы более простым.
И по возможности лучше разрабатывать код таким образом,
чтобы не было глобального разделяемого ресурса, или состояния.

In [ ]:

```
# Синхронизация потоков, race condition

import threading

class Point(object):
    def __init__(self, x, y):
        self.set(x, y)

    def get(self):
        return (self.x, self.y)

    def set(self, x, y):
        self.x = x
        self.y = y

# use in threads
my_point = Point(10, 20)
my_point.set(15, 10)
my_point.get()
```

In [ ]:

```
# Синхронизация потоков, блокировки

import threading

class Point(object):
    def __init__(self, x, y):
        self.mutex = threading.RLock()
        self.set(x, y)

    def get(self):
        with self.mutex:
            return (self.x, self.y)

    def set(self, x, y):
        with self.mutex:
            self.x = x
            self.y = y

# use in threads
my_point = Point(10, 20)
my_point.set(15, 10)
my_point.get()
```

In [ ]:

Этот код гарантирует что если объект класса Point будет использоваться в разных потоках,
то изменение x и y будет всегда атомарным.

Работает все это так: - при вызове метода берем блокировку 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 атомарный объект.
Если позволить одному потоку поменять x, а другой в это же время поправит y
логика алгоритма может сломаться.

In [ ]:

```
# Синхронизация потоков, блокировки

import threading


a = threading.RLock()
b = threading.RLock()

def foo():
    try:
        a.acquire()
        b.acquire()
    finally:
        a.release()
        b.release()
```

In [ ]:
```
# Синхронизация потоков, условные переменные

class Queue(object):
    def __init__(self, size=5):
        self._size = size
        self._queue = []
        self._mutex = threading.RLock()
        self._empty = threading.Condition(self._mutex)
        self._full = threading.Condition(self._mutex)

    def put(self, val):
        with self._full:
            while len(self._queue) >= self._size:
                self._full.wait()

            self._queue.append(val)
            self._empty.notify()

    def get(self):
        with self._empty:
            while len(self._queue) == 0:
                self._empty.wait()

            ret = self._queue.pop(0)
            self._full.notify()
            return ret
```

In [ ]:
```
Все механизмы блокировки и обмена данными между потоками
имеют место и для процессов.
Но вместо модуля threading нужно использовать multiprocessing
.
```

Глобальная блокировка интерпретатора, GIL

- Что такое Global Interpreter Lock?
- Зачем нужен GIL?
- GIL и системные вызовы

In [ ]:

GIL это достаточно сложная тема в Python.
Для более глубокого понимания того как работают потоки
нужно иметь общее представление зачем нужен GIL и как он устр
оен.

GIL защищает память интерпретатора от повреждений и делает оп
ерации атомарными.

Поток, владеющий GIL, не отдает его пока об этом не попросят.
Потоки засыпают на 5 мс. для ожидания GIL.
Сам GIL устроен как обычная нерекурсивная блокировка. Эта же
структура лежит в основе threading.Lock.

Когда Python делает системный вызов или вызов из внешней библ
иотеки он отключает механизм GIL.
После того как функция вернет управление снова включает его.

Т.е. потоки при своем выполнении так или иначе вынуждены полу
чать GIL.
Именно поэтому многопоточные программы, требующие больших выч
ислений,
могут выполняться медленней чем однопоточные.

In [ ]:

```
# cpu bound programm

from threading import Thread
import time

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

# series run
t0 = time.time()
count(100_000_000)
count(100_000_000)
print(time.time() - t0)

# parallel run
t0 = time.time()
th1 = Thread(target=count, args=(100_000_000,))
th2 = Thread(target=count, args=(100_000_000,))

th1.start(); th2.start()
th1.join(); th2.join()
print(time.time() - t0)
```

In [ ]:

```
# как выполняется поток?

a      r      a            r              a          r    a
  run  |------|    run     |--------------|   run    |----| r
un
------>|  IO  |----------->|      IO      |--------->| IO |--
--->
       |------|            |--------------|          |----|
a      r      a            r              a          r    a

a - acquire GIL
r - release GIL
```