Многопоточное программирование

Процесс?

Процесс – программа, которая запущена в ОП компьютера.

- Характеристики процесса:

1. Идентификатор процесса PID
2. Объем ОП
3. Стек (стек используется для вызова функций, для создания локальных переменных этих функций)
4. Список открытых файлов
5. Ввод/вывод

```
import time
import os

pid = os.getpid()

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

```python
import time
import os

pid = os.fork() #создает точную копию родит процесса
#то есть все ресурсы родит процесса будут скопированы в дочерний
#То есть после того, как системный вызов fork отработал, с этого момента у нас два процесса в операционной системе.
#Единственное отличие заключается в том, что системный вызов fork в родительский процесс вернет pid дочернего процесса,
#а в дочернем процессе переменная pid будет равна нулю.

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

```
import time
import os

a = "a"

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

```
import time
import os

f = open("log.txt")
a = f.readline()

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

#память копируется из родит в дочерний
```

```python
import time
import os
from multiprocessing import Process

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


p = Process(target=f, args=("Bob",))
#передаем в конструктор функцию которую хотим исполнить
#и ее аргументы
p.start() #создаем процесс у объекта – будет вызван системный вызов форк и исполнена функция f в отдельном процессе
p.join() #ожидание завершения всех дочерних процессов

#память копируется из родит в дочерний
```

```python
import time
import os
from multiprocessing import Process
#создание процесса при помощи наследования

class PrintProcess(Process):
    def __init__(self, name):
        super().__init__()
        self.name = name
    def run(self):
        print("hello", self.name)


p = PrintProcess("Bob")

p.start()
p.join()
```

ПОТОКИ

- Поток напоминает процесс
- У потока своя последовательность инструкции
- Каждый поток имеет собственный стек
- Все потоки выполняются в рамках процесса(делят его память и ресурсы)
- Управлением выполнением потоков занимается ОС
- Потоки в питоне имеют свои ограничения

Много поточность

Способность процесса выполнять несколько потоков параллельно, называется многопоточностью.

Плюсы	Минусы
1. Повышает скорость вычислений (каждое ядро или процессор обрабатывает отдельный поток одновременно) 2. Позволяет программе оставаться отзывчивым в то время как один поток ожидает ввода, в другой работает граф интерфейс 3. Поток имеет глобальные и локальные переменные	1. На однопроцессорной системе много поточность не влияет на скорость вычислений 2. Синхронизация, чтобы избежать взаимного исключения при доступе к общим ресурсам (загрузка процессора) 3. Увеличивается сложность программы

Python предлагает два модуля для реализации threads в программах.

- модуль <thread>
- модуль <threading>.

модуль <thread> является устаревшим в Python 3 и переименован в модуль <_thread> для обратной совместимости.

Основное различие между этими двумя модулями является то, что модуль <thread> реализует нить как функцию. С другой стороны, модуль <threading> предлагает объектно-ориентированный подход для обеспечения возможности создания потоков.

Модуль <threading> также представляет класс <Thread>

<Thread> имеет следующие методы:

Методы класса	Метод Описание
run():	Это функция точки входа для любого потока.
start():	Способ **start**() запускает поток, когда вызывается метод run.
join([time]):	Метод **join**() позволяет программе ожидать оканчиваются.
isAlive():	Метод **isAlive**() проверяет активную нить.
getName():	Метод **getName**() возвращает имя потока.
setName():	Метод **setName**() обновляет имя потока.

```
import time
import os
from threading import Thread
def f(name):
    print("hello", name)

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

Создание класса потоков и объектов для печати текущей даты

```
import threading
import datetime

class myThread (threading.Thread):
    def __init__(self, name, counter):
        threading.Thread.__init__(self)
        self.threadID = counter
        self.name = name
        self.counter = counter
    def run(self):
        print("Запуск "+self.name)
        print_date(self.name, self.counter)
        print("Выход "+self.name)

def print_date(threadName, counter):
    date = datetime.datetime.now()
    print(f"{threadName}[{counter}]: {date}")

thread1 = myThread("Поток", 1)
thread2 = myThread("Поток", 2)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Выход")
```

Модуль <threading> имеет встроенные функциональные возможности для **осуществления блокировки,** что позволяет синхронизировать потоки. Блокировка требуется для управления доступом к общим ресурсам для предотвращения повреждения или непринятых данных.

Вы можете вызвать метод Lock(), чтобы применять блокировки, он возвращает новый объект блокировки. Затем, **вы можете вызвать метод acquire(blocking)** объекта блокировки для обеспечения синхронного выполнения потоков.

```
import threading
import datetime

class myThread (threading.Thread):
    def __init__(self, name, counter):
        threading.Thread.__init__(self)
        self.threadID = counter
        self.name = name
        self.counter = counter
    def run(self):
        print("Запуск "+self.name)
        threadLock.acquire()
        print_date(self.name, self.counter)
        threadLock.release()
        print("Выход "+self.name)

def print_date(threadName, counter):
    date = datetime.datetime.now()
    print(f"{threadName}[{counter}]: {date}")

threadLock = threading.Lock()

thread1 = myThread("Поток", 1)
thread2 = myThread("Поток", 2)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Выход")
```

Очередь

```
import time
import os
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) #объект очередь с макс размером 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) #помещение элементов в очередь
    #(если в очереди уже 5, то ждет пока не появится свободное место)

q.put(None) #для корректного завершения работы потока
q.put(None)
th1.join()
th2.join()
```

Поиск слов в файлах

```python
import threading
from queue import Queue

class Worker(threading.Thread):
    def __init__(self, work_queue, word):
        threading.Thread.__init__(self)
        self.work_queue = work_queue
        self.word = word

    def run(self):
        try:
            filename = self.work_queue.get()
            self.process(filename)
        finally:
            pass

    def process(self, filename):
        previous = ""
        current = True
        with open(filename, "rb") as fh:
            while current:
                current = fh.readline()
                if not current: break
                current = current.decode("utf8")
                if self.word in current:
                    print("find {0}: {1}".format(self.word, filename))
                previous = current


word = 'import'
filelist = ['./file1.py', './file2.py', './file3.py']
work_queue = Queue()
for filename in filelist:
    work_queue.put(filename)
for i in range(3):
    worker = Worker(work_queue, word)
    worker.start()
```

```python
import os
import threading

#синхронизация поток, 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()
            #notify() Выводит из режима ожидания один из потоков, ожидающих данные

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

            ret = self._queue.pop(0)
            self._full.notify()
            return ret
# Условные переменные в конструктор получает объект блокировки.
#  Он есть по умолчанию, но если у нас эти переменные взаимозависимые,
#  то необходимо использовать общую блокировку.
```

Класс RLock - вариант простой блокировки, которая блокирует поток только в том случае, если блокировка захвачены другим потоком

если требуется использовать несколько условных переменных для синхронизации доступа к одному ресурсу.

Семафоры

Семафоры - более сложный и совершенный механизм блокировок. Внутри семафора - счетчик, в отличии от объекта блокировки, в которой просто флажок.

Семафор блокирует поток только когда более заданного числа потоков пытаются захватить семафор.

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

```
max_connections = 10
semaphore = threading.BoundedSemaphore(max_connections)
```

GIL

- GIL – глобальная блокировка интерпретатора(GLOBAL INTERPRETER LOCK)

Эта штука позволяет одновременно запускать только **один питоновский поток** — остальные обязаны ждать переключения GIL на них.

```
from threading import Thread
import time

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

t0 = time.time()
count(100_000_000)#уменьшает до 0
count(100_000_000)
print(time.time() - t0)

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)

#работа с потоками и без
```

У нас есть поток, в котором выполняется наш Python код, и каждый раз Python интерпретатор пробует получить глобальную блокировку интерпретатора.

- 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 библиотеки он отключает механизм GIL.
После того как функция вернет управление снова включает его.

Т.е. потоки при своем выполнении так или иначе вынуждены получать GIL.

Именно поэтому многопоточные программы, требующие больших вычислений,
могут выполняться медленней чем однопоточные.