

Multiprocessing is All you Need

(Немного хайпово звучит)

```
In [1]: import os
import time
from multiprocessing import Process
```

Делаем подпроцессы, которые что-то делают

```
In [2]: def doubler(number):  
        result = number * 2  
        proc = os.getpid()  
        print(  
            '{0} doubled to {1} by process id: {2}'.format(  
                number, result, proc))  
        return result  
  
numbers = [5, 10, 15, 20, 25]  
procs = []  
  
for index, number in enumerate(numbers):  
    proc = Process(target=doubler, args=(number,))  
    procs.append(proc)  
    proc.start()  
  
for proc in procs:  
    proc.join()
```

5 doubled to 10 by process id: 5581

10 doubled to 20 by process id: 5582

15 doubled to 30 by process id: 558920 doubled to 40 by process id: 5592

25 doubled to 50 by process id: 5597

Теперь можем задавать еще и имена подпроцессов (скорее нужно для логирования)

```
In [3]: from multiprocessing import current_process
```

```
In [4]: def doubler(number):
        result = number * 2
        proc_name = current_process().name
        print('{0} doubled to {1} by: {2}'.format(
            number, result, proc_name))

        numbers = [5, 10, 15, 20, 25]
        procs = []
        proc = Process(target=doubler, args=(5,))

        for index, number in enumerate(numbers):
            proc = Process(target=doubler, args=(number,))
            procs.append(proc)
            proc.start()

        proc = Process(target=doubler, name='Test', args=(2,))
        proc.start()
        procs.append(proc)

        for proc in procs:
            proc.join()
```

```
5 doubled to 10 by: Process-7
15 doubled to 30 by: Process-910 doubled to 20 by: Process-820 doubled to 40 b
y: Process-10
```

```
2 doubled to 4 by: Test
25 doubled to 50 by: Process-11
```

Большой минус предыдущих примеров состоит в том, что по умолчанию `Process` не умеет ничего возвращать (типо не функция а процедура). Хм, а почему бы не сделать просто глобальную переменную?


```
In [5]: results = dict()

def doubler(number, results):
    result = number * 2
    proc = os.getpid()
    results[proc] = result
    print(
        '{0} doubled to {1} by process id: {2}'.format(
            number, result, proc))

numbers = [5, 10, 15, 20, 25]
procs = []

for index, number in enumerate(numbers):
    proc = Process(target=doubler, args=(number, results))
    procs.append(proc)
    proc.start()

for proc in procs:
    proc.join()

print(results)
```

```
5 doubled to 10 by process id: 5636
10 doubled to 20 by process id: 5637
15 doubled to 30 by process id: 5644
20 doubled to 40 by process id: 5645
```

```
25 doubled to 50 by process id: 5650
{}
```

Хм не вышло... Почему? Потому что на самом деле создается отдельная программа, которая выполняет данный кусок кода (аналог `fork` в си). По простому: создается полный дубликат процесса в другом куске оперативной памяти. Как следствие имеем несколько процессов, теперь нужно научить их взаимодействовать.

Для этого можно создать менеджер сообщений между процессами.

```
In [6]: from multiprocessing import Manager
```

```
In [7]: manager = Manager() # здесь есть вариативность: Queue, Pipe  
results = manager.dict()
```

```
def doubler(number, results):  
    result = number * 2  
    proc = os.getpid()  
    results[proc] = result  
    print(  
        '{0} doubled to {1} by process id: {2}'.format(  
            number, result, proc))  
  
numbers = [5, 10, 15, 20, 25]  
procs = []  
  
for index, number in enumerate(numbers):  
    proc = Process(target=doubler, args=(number, results))  
    procs.append(proc)  
    proc.start()  
  
for proc in procs:  
    proc.join()  
  
print(results)
```

```
5 doubled to 10 by process id: 566610 doubled to 20 by process id: 5669
```

```
20 doubled to 40 by process id: 5680
```

```
15 doubled to 30 by process id: 5674
```

```
25 doubled to 50 by process id: 5686
```

```
{5666: 10, 5669: 20, 5674: 30, 5680: 40, 5686: 50}
```

Один из прикольных вариантов это создать массив, который шариться между процессами

```
In [8]: from multiprocessing import Array
```

```
In [9]: def doubler(index, arr):
        arr[index] = arr[index]*2

        numbers = [5, 10, 15, 20, 25]
        procs = []

        arr = Array('d', numbers)
        print(arr[:])
        for index, number in enumerate(numbers):
            proc = Process(target=doubler, args=(index, arr))
            procs.append(proc)
            proc.start()

        for proc in procs:
            proc.join()

        print(arr[:])
```

```
[5.0, 10.0, 15.0, 20.0, 25.0]
[10.0, 20.0, 30.0, 40.0, 50.0]
```


На самом деле все что было до этого это не интересно. Это все просто похоже на потоки (но не потоки а процессы). Да и каждый раз что-то думать с расшареной памятью, чтобы все работало корректно(кто когда-то паралелил на си, тот поймет). Обычно в жизни часто используется метод Pool.

```
In [10]: from multiprocessing import Pool
```

```
In [11]: def doubler(number):  
          return number * 2  
  
          numbers = [5, 10, 20]  
          pool = Pool(processes=3)  
          for item in pool.map(doubler, numbers):  
              print(item)
```

```
10  
20  
40
```

В данном случае процессы выполняются параллельно, но обязательно выдаются последовательно. Давайте в этом убедимся.

In [12]:

```
%%time
def doubler(number):
    time.sleep(3)
    return number * 2

numbers = [5, 10, 20]
pool = Pool(processes=3)
for item in pool.map(doubler, numbers):
    print(item)
```

10

20

40

CPU times: user 13.1 ms, sys: 5.32 ms, total: 18.5 ms

Wall time: 3.01 s

Замечание: `pool.map` всегда возвращает готовый массив! Чтобы постепенно получать результаты нужно пользоваться `pool.imap` !!!

In [13]:

```
%%time
def doubler(number):
    time.sleep(number)
    return number * 2

numbers = [1, 2, 3]
pool = Pool(processes=3)
for item in pool.map(doubler, numbers):
    print(item, time.ctime())

2 Fri Dec 11 10:15:34 2020
4 Fri Dec 11 10:15:34 2020
6 Fri Dec 11 10:15:34 2020
CPU times: user 13.6 ms, sys: 9.7 ms, total: 23.3 ms
Wall time: 3.02 s
```

In [14]:

```
%%time
def doubler(number):
    time.sleep(number)
    return number * 2

numbers = [1, 2, 3]
pool = Pool(processes=3)
for item in pool.imap(doubler, numbers):
    print(item, time.ctime())

2 Fri Dec 11 10:15:35 2020
4 Fri Dec 11 10:15:36 2020
6 Fri Dec 11 10:15:37 2020
CPU times: user 20.4 ms, sys: 10.6 ms, total: 31 ms
Wall time: 3.01 s
```


Представим ситуацию, что у функция выполняется разное количество времени на разных процессах. К примеру получается так, что первый вход выполняется дольше всего (например первые входные данные выполняются в 2 раза дольше чем все другие), тогда `pool.imap` застрнет и будет ждать первый (пока остальные будут толпится в очереди). Хочется получить что-то типо <<Первый обработался, первый вернуля>>. Для этого есть специальное расширение `pool.imap_unordered`. Который возвращает в порядке готовности.

In [15]:

```
%%time
def doubler(number):
    time.sleep(3-number)
    return number * 2

numbers = [1, 2, 3]
pool = Pool(processes=3)
for item in pool.imap_unordered(doubler, numbers):
    print(item, time.ctime())

6 Fri Dec 11 10:15:37 2020
4 Fri Dec 11 10:15:38 2020
2 Fri Dec 11 10:15:39 2020
CPU times: user 12.3 ms, sys: 18 ms, total: 30.3 ms
Wall time: 2.01 s
```

Вывод

- Для обработки большого количества данных в одном формате пользуемся `pool` (ясное дело, если обработка каждой части не зависима).
 - В случае, если последовательность входа и выхода важна, то пользуемся `pool.imap`.
 - В случае, если последовательность входа не важна, то пользуемся `pool.imap_unordered`.
 - Пользоваться `pool.map` не рекомендуется, так как забивается память.
- Рассмотрим простой пример. Пусть мы хотим обработать текст (сделать lower case). Вход это текст и выход это текст. Пусть текстов очень много (500 гб). Выполняя `pool.imap` мы сможем постепенно забирать тексты с памяти и передавать дальше (ну к примеру сохранять на диск) в то время как `pool.map` будет заблокирован пока не обработает все файлы (во первых процесс заложен и файлы далее по конвейеру не пойдут, а во вторых забивается оперативная память).