Multiprocessing In Python

An Introduction to Multi Processing in Python with multiprocessing module

Here's what we'll cover :

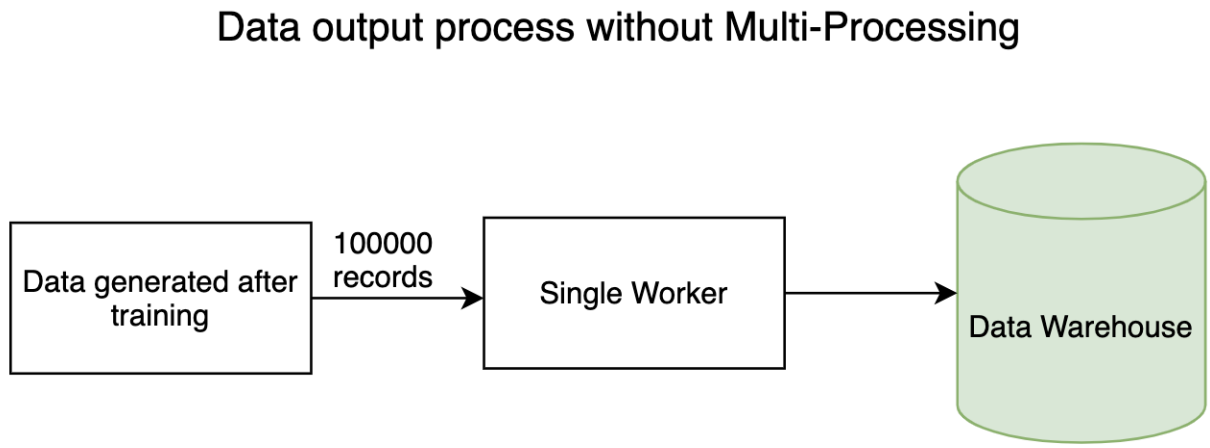
- **What is Multiprocessing**
- **multiprocessing module**
- **The four most important classes of multiprocessing module**
- **Difference between Multi Threading and Multi Processing**

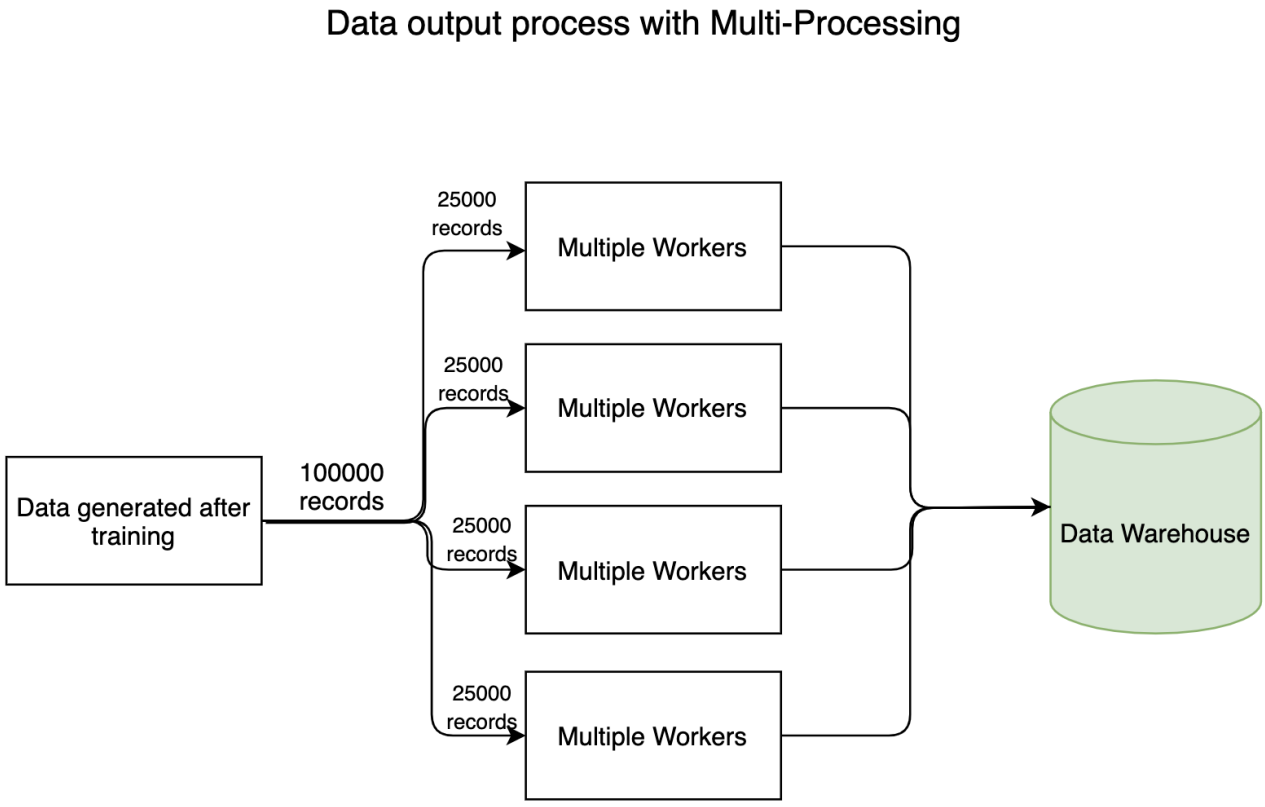
What is MultiProcessing ?

- **It is the process based multi tasking which is a parallelism**
- **Multiprocessing is a mode of operation in which two or more processors in a computer simultaneously process two or more different portions of the same program (set of instructions).**
- **Multiprocessing allows you to spawn multiple processes within a program**
- **It allows you to leverage multiple CPU cores on your machine, Since multiple processes are involved, the time taken for doing the job improves greatly with the tradeoff of your system resources.**

- **Side steps the GIL limitation of Python which allows only one thread to hold control of the Python interpreter**

- **Used for computation or CPU intensive programs**





multiprocessing In Python:

- **In python, multiprocessing is natively supported via multiprocessing which is a package that supports spawning processes using an API similar to the threading module**

- **The multiprocessing package offers both local and remote parallelism, effectively side-stepping the Global Interpreter Lock by using sub-processes instead of threads**

Understanding Multiprocesser

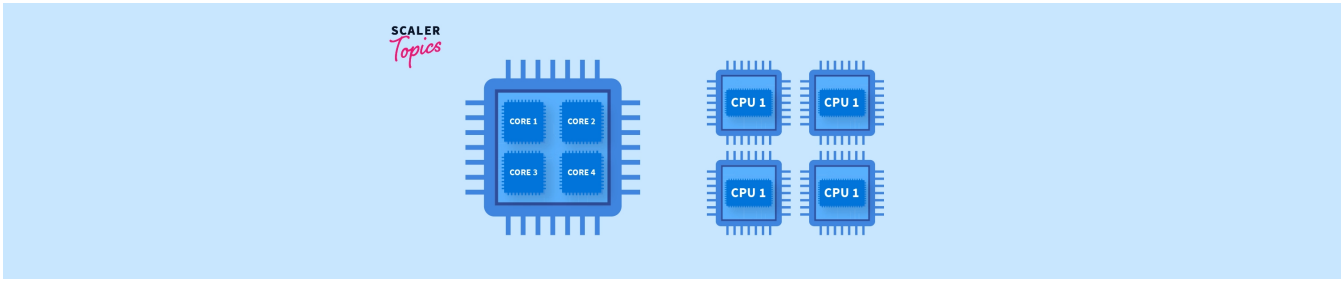
- **A multiprocessor- a computer with more than one central processor.**

- **A multi-core processor- a single computing component with more than one independent actual processing units/ cores.**

- **If a computer has only one processor with multiple cores, the tasks can be run parallel using multithreading**

- **A multiprocessor system has the ability to support more than one processor at the same time.**

- **To find the number of CPU cores available on our system, we use mp.cpu_count() function.**



In [1]:
```
import multiprocessing as mp

print(mp.cpu_count())
```

```
8
```

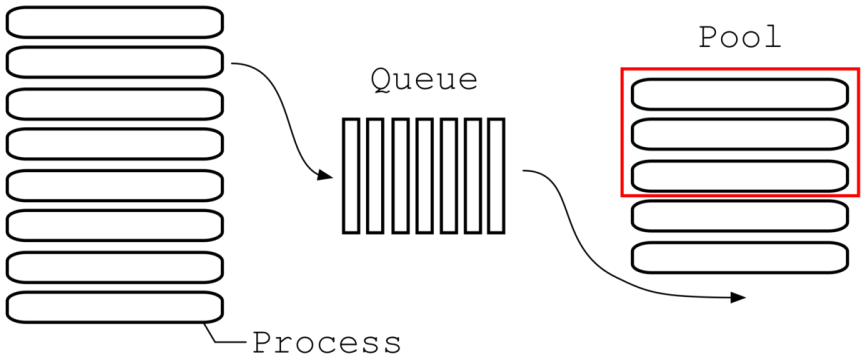
The four most important classes of multiprocessing module :

- **Process Class**

- **Lock Class**

- **Queue Class**

- **Pool Class**



Process Class :

- **When we work with Multiprocessing,at first we create process object**

- **Process is the forked copy of the current process. It creates a new process identifier and tasks run as independent child process**

- **start ( ) and join ( ) functions belong to this class. To pass an argument through a process, we use args keyword.**

In [2]:
```
import multiprocessing
import time
def calc_square(numbers):
    for n in numbers:
        time.sleep(1)
        print('square ' + str(n*n))

def calc_cube(numbers):
    for n in numbers:
        time.sleep(1)
        print('cube '+ str(n*n*n))
```

In [3]:
```
#Without Multiprocessing
arr=[2,3,8,9]

tic = time.time()
calc_square(arr)
calc_cube(arr)
toc = time.time()
print('Done in {:.4f} seconds'.format(toc-tic))
```

```
square 4
square 9
square 64
```

```
square 81
cube 8
cube 27
cube 512
cube 729
Done in 8.0677 seconds
```

In [4]:
```
#Multiprocessing
arr=[2,3,8,9]
tic = time.time()

#Creating new  process identifier
p1=multiprocessing.Process(target=calc_square,args=(arr,))
p2=multiprocessing.Process(target=calc_cube,args=(arr,))

#Starting the process
p1.start()
p2.start()

#order to execute the rest of the program after the multi-process functions are executed
p1.join()
p2.join()


toc = time.time()
print('Done in {:.4f} seconds'.format(toc-tic))
```

```
Done in 0.1790 seconds
```

Queue Class :

- **Queue is a data structure which uses First In First Out (FIFO) technique**
- **It helps us perform inter process communication using native Python objects.**
- **Queue enables the Process to consume shared data when passed as a parameter.**
- **put ( ) function is used to insert data to the queue and get ( ) function is used to consume data from the queue.**

In [5]:
```
from multiprocessing import Queue

colors = ['red', 'green', 'blue', 'black']
cnt = 1
# instantiating a queue object
queue = Queue()
print('pushing items to queue:')
for color in colors:
    print('item no: ', cnt, ' ', color)
    queue.put(color)
    cnt += 1

print('\npopping items from queue:')
cnt = 0
```

```
while not queue.empty():
    print('item no: ', cnt, ' ', queue.get())
    cnt += 1
```

```
pushing items to queue:
item no:  1   red
item no:  2   green
item no:  3   blue
item no:  4   black

popping items from queue:
item no:  0   red
item no:  1   green
item no:  2   blue
item no:  3   black
```

In [ ]:
```
#Excecute in VS Code
import multiprocessing

def even_no(num, n):
    for i in num:
        if i % 2 == 0:
            n.put(i)

if __name__ == "__main__":
    n = multiprocessing.Queue()
    p = multiprocessing.Process(target=even_no, args=(range(10), n))
    p.start()
    p.join()

    while n:
        print(n.get())
```

Lock Class :

- **The lock class allows the code to be locked in order to make sure that no other process can execute the similar code until it is released**

- **To claim the lock, acquire ( ) function is used and to release the lock, release ( ) function is used.**

In [6]:
```
#Multiprocessing Lock for ATM Functionality
import multiprocessing

# Withdrawal function

def wthdrw(bal, lock):
    for _ in range(10000):
        lock.acquire()
        bal.value = bal.value - 1
        lock.release()

# Deposit function

def dpst(bal, lock):
    for _ in range(10000):
        lock.acquire()
        bal.value = bal.value + 1
```

```
        lock.release()

def transact():
    # initial balance
    bal = multiprocessing.Value('i', 100)
    # creating lock object
    lock = multiprocessing.Lock()
    # creating processes
    proc1 = multiprocessing.Process(target=wthdrw, args=(bal,lock))
    proc2 = multiprocessing.Process(target=dpst, args=(bal,lock))
    # starting processes
    proc1.start()
    proc2.start()
    # waiting for processes to finish
    proc1.join()
    proc2.join()
    # printing final balance
    print("Final balance = {}".format(bal.value))

if __name__ == "__main__":
    for _ in range(10):
    # performing transaction process
        transact()
```

```
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
```

Pool Class :

- **The pool class helps us execute a function against multiple input values in parallel.**
- **This concept is called Data Parallelism.**
- **pool.map() function is used to pass a list of multiple arguments.**

In [ ]:

```
#Excecute in VS Code
from multiprocessing import Pool
import time

work = (["A", 5], ["B", 2], ["C", 1], ["D", 3])

def work_log(work_data):
    print(" Process %s waiting %s seconds" % (work_data[0], work_data[1]))
    time.sleep(int(work_data[1]))
    print(" Process %s Finished." % work_data[0])


def pool_handler():
    p = Pool(2)
    p.map(work_log, work)
```

```
if __name__ == '__main__':
    pool_handler()
```

Python ProcessPoolExecutor :

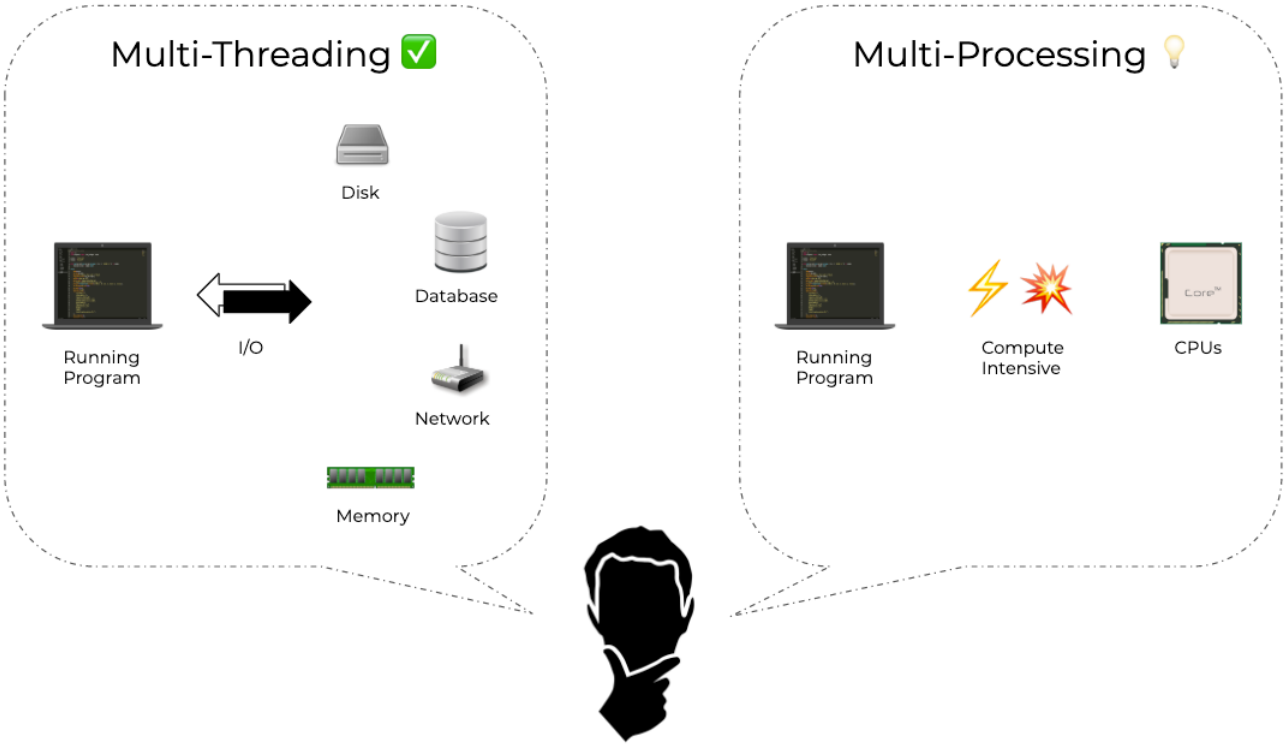
- **The ProcessPoolExecutor Python class is used to create and manage process pools and is provided in the concurrent.futures module.**

- **The ProcessPoolExecutor extends the Executor class and will return Future objects when it is called.**

In [7]:
```
import time
import os
from concurrent.futures import ProcessPoolExecutor
from PIL import Image, ImageFilter

filenames = [
    'images/1.jpg',
    'images/2.jpg',
    'images/3.jpg',
    'images/4.jpg',
    'images/5.jpg',
]

def create_thumbnail(filename, size=(50,50),thumb_dir ='thumbs'):
    img = Image.open(filename)
    img = img.filter(ImageFilter.GaussianBlur())
    img.thumbnail(size)
    img.save(f'{thumb_dir}/{os.path.basename(filename)}')
    print(f'{filename} was processed...')


if __name__ == '__main__':
    start = time.perf_counter()
    with ProcessPoolExecutor() as executor:
        executor.map(create_thumbnail, filenames)
    finish = time.perf_counter()
    print(f'It took {finish-start: .2f} second(s) to finish')
```

```
It took  0.25 second(s) to finish
```

Multiprocessing Vs Multitherding



TLDR : Multithreading for IO-bound tasks. Multiprocessing for CPU-bound tasks.

In [8]:
```
import time, os
from threading import Thread, current_thread
from multiprocessing import Process, current_process
COUNT = 200000000
SLEEP = 10
def io_bound(sec):
    pid = os.getpid()
    threadName = current_thread().name
    processName = current_process().name
    print(f"{pid} * {processName} * {threadName} \
        ---> Start sleeping...")
    time.sleep(sec)
    print(f"{pid} * {processName} * {threadName} \
        ---> Finished sleeping...")
def cpu_bound(n):
    pid = os.getpid()
    threadName = current_thread().name
    processName = current_process().name
    print(f"{pid} * {processName} * {threadName} \
        ---> Start counting...")
    while n>0:
        n -= 1
    print(f"{pid} * {processName} * {threadName} \
        ---> Finished counting...")
```

In [9]:
```
#IO Task Without Threading
if __name__=="__main__":
    start = time.time()
    io_bound(SLEEP)
    io_bound(SLEEP)
    end = time.time()
    print('Time taken in seconds -', end - start)
```

```
12576 * MainProcess * MainThread         ---> Start sleeping...
12576 * MainProcess * MainThread         ---> Finished sleeping...
12576 * MainProcess * MainThread         ---> Start sleeping...
12576 * MainProcess * MainThread         ---> Finished sleeping...
Time taken in seconds - 20.001720666885376
```

In [10]:
```
#IO Task With Threading
if __name__=="__main__":
    start = time.time()
    t1 = Thread(target=io_bound, args=(SLEEP,))
    t2 = Thread(target=io_bound, args=(SLEEP,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    end = time.time()
    print('Time taken in seconds -', end - start)
```

```
12576 * MainProcess * Thread-6         ---> Start sleeping...
12576 * MainProcess * Thread-7         ---> Start sleeping...
12576 * MainProcess * Thread-7         ---> Finished sleeping...12576 * MainProcess * Th
read-6         ---> Finished sleeping...

Time taken in seconds - 10.01404857635498
```

In [11]:
```
#IO Task With Multiprocessing
if __name__=="__main__":
    p1 = Process(target=io_bound, args=(SLEEP,))
    p2 = Process(target=io_bound, args=(SLEEP,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print('Time taken in seconds -', end - start)
```

```
Time taken in seconds - 10.01404857635498
```

In [12]:
```
#CPU Bound Task Without Threading
if __name__=="__main__":
    start = time.time()
    cpu_bound(COUNT)
    cpu_bound(COUNT)
    end = time.time()
    print('Time taken in seconds -', end - start)
```

```
12576 * MainProcess * MainThread         ---> Start counting...
12576 * MainProcess * MainThread         ---> Finished counting...
12576 * MainProcess * MainThread         ---> Start counting...
12576 * MainProcess * MainThread         ---> Finished counting...
Time taken in seconds - 77.20385670661926
```

In [13]:
```
#CPU Bound Task With Threading
if __name__=="__main__":
    start = time.time()
    t1 = Thread(target=cpu_bound, args=(COUNT,))
    t2 = Thread(target=cpu_bound, args=(COUNT,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    end = time.time()
    print('Time taken in seconds -', end - start)
```

```
12576 * MainProcess * Thread-8         ---> Start counting...
12576 * MainProcess * Thread-9         ---> Start counting...
12576 * MainProcess * Thread-9         ---> Finished counting...
12576 * MainProcess * Thread-8         ---> Finished counting...
Time taken in seconds - 71.44468235969543
```

In [14]:
```
#CPU Bound Task With Multiprocessing
if __name__=="__main__":
    start = time.time()
```

```
    p1 = Process(target=cpu_bound, args=(COUNT,))
    p2 = Process(target=cpu_bound, args=(COUNT,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    end = time.time()
    print('Time taken in seconds -', end - start)
```

```
Time taken in seconds - 0.18386363983154297
```

Major Diffrences

FEATURE	THREADING	MULTIPROCESSING
Short Description	Multiple threads of a single process executed concurrently	Multiple processes executed concurrently
Aim	Enable applications to be responsive	Multi-task Heavy lifting
Memory Type	Shared Memory	Isolated Memory
Memory Footprint/usage	Lightweight	Heavyweight
GIL	GIL limits execution to one thread	No GIL Limitations
Children Processes	Non-interruptible/Non-killable	Interruptible/Killable
Code Readability	Complicated	Straightforward
Recommended task	I/O-bound apps	CPU-bound apps
Examples	webscrapping, database queries	ML algo training

FAQ's on Multiprocessing :

- **What is Python multiprocessing module ?**
- **How does multiprocessing work in Python ?**
- **How to stop multiprocessing in Python ?**
- **Explain the purpose for using multiprocessing module in Python.**
- **Is multiprocessing faster than multithreading in Python ?**