Python Multithreading vs Multiprocessing: -

Use multiprocessing when you have to cpu intensive task and would take benefit of multiple core (more core of cpu) and avoid the global interpreter lock. ( effectively side-stepping the [Global Interpreter Lock](https://docs.python.org/3/glossary.html#term-global-interpreter-lock) by using subprocesses instead of threads.)

  Set deamon - True to create daemon process in threading

Multithreading thread are **light weight processes**, they require less memory overhead and hence are cheaper than processes.

* The **register set** and **local variables** of each threads are stored in the stack.
* The **global variables** (stored in the heap) and the **program codes** are shared among all the threads

Use multithreading when you have a task of multiple I/O or network usage.

How to stop thread that is running even timeout defined in join exceeds? Using Event

from threading import Thread

from threading import Event

import time

class Connection(Thread):

   StopEvent = 0

   def \_\_init\_\_(self,args):

       Thread.\_\_init\_\_(self)

       self.StopEvent = args

   def run(self):

       for i in range(1,10):

           if(self.StopEvent.wait(0)):

               print('Asked to stop')

               break;

           print(f'Child process is sleeping with count {i}')

           time.sleep(3)

       print('Child Thread is exiting')

stop = Event()

conn = Connection(stop)

conn.start()

print('Main Thread is start to wait for 5 sec for Child Thread')

conn.join(5)

print('Main Thread cant wait for more then 5 sec and now stopping the child thread')

stop.set()

conn.join()

print('Main Thread is exiting')

We can **never** set active thread to daemon or non-daemon (vice-versa as well)

If we want to change the **main** thread which is always non-daemon in nature to daemon nature then we will get a ***RuntimeError*** because when the program is started at a time main thread is also started so the main thread is an active thread and the active thread is not set to the daemon.

[multiprocessing](https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing) supports two types of communication channel between processes:

* Queue
* Pipe

The [Pipe()](https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Pipe) function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

from multiprocessing import Process, Pipe

def f(conn):

    conn.send([42, None, 'hello'])

    conn.close()

if \_\_name\_\_ == '\_\_main\_\_':

    parent\_conn, child\_conn = Pipe()

    p = Process(target=f, args=(child\_conn,))

    p.start()

    print(parent\_conn.recv())   # prints "[42, None, 'hello']"

    p.join()

 Synchronization between processes : Using lock to ensure that diffentent thread/process are not writing/doing similar operation at same time.

Sharing state with processes:-

* Shared memory
* Server Process (Manager object)

from multiprocessing import Process, Manager  
  
def f(d,l):  
 d['name'] = 'MP'  
 d['purpose'] = 'Shared processs data'  
 d['written'] = 'Programmer'  
 l.reverse()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with Manager() as manager:  
 d = manager.dict()  
 l = manager.list(range(10))  
  
 p = Process(target=f, args=(d,l))  
 p.start()  
 p.join()  
  
 print(d)  
 print(l)

Pool Vs Process: -

Management: -

The Pool class is easier to use than the Process class because you do not have to manage the processes by yourself. It creates the processes, splits the input data, and returns the result in a list. It also waits for the workers to finish their tasks, i.e., you do not have to call the join() method explicitly.

Memory: -

**While the Process keeps all the processes in the memory, the Pool keeps only those that are under execution. Therefore, if you have a large number of tasks, and if they have more data and take a lot of space too, then using process class might waste a lot of memory**.

The overhead of creating a Pool is more. Therefore, when there are a small number of tasks, and they are not repetitive, it is advisable to use a Process in this case.

I/O Operation: -

Both the Process and the Pool class use FIFO (First In First Out) scheduler. However, **if the current process is waiting for, or executing an I/O operation, then the Process class halts the current one and schedules another one from the task queue. The Pool class, on the other hand, waits for the process to complete its I/O operation**, i.e., it does not schedule another one until the current has finished its execution. Because of this, the execution time might increase. **Process is preferred over Pool when your task is I/O bound** (A program is I/O bound if it spends most of its time waiting for the I/O operation to complete).

import multiprocessing as mp  
import time  
  
def test(fname):  
 with open(fname,"w") as f:  
 f.write("Hi")  
 f.write("Hi 1")  
 f.write("Hi 2")  
 f.write("Hi 3")  
 f.write("Hi 4")  
  
  
def process\_performance(filename):  
 start\_time = time.time()  
 # filename = "text.txt"  
 p1 = mp.Process(target=test, args=(filename,))  
 p2 = mp.Process(target=test, args=("text-2.txt",))  
  
 p1.start()  
 p2.start()  
 p1.join()  
 p2.join()  
 end\_time = time.time()  
 print(f"Process :: I/O :: Total time taken {end\_time - start\_time}")  
  
  
def pool\_performance(filename):  
 start\_time = time.time()  
 pool = mp.Pool()  
 a = pool.apply\_async(test, args=(filename,))  
 b = pool.apply\_async(test, args=("text-2.txt",))  
 a.wait()  
 b.wait()  
 end\_time = time.time()  
 print(f"Pool :: I/O :: Total time taken {end\_time - start\_time}")  
  
  
if \_\_name\_\_ == "\_\_main\_\_" :  
 filename = "text.txt"  
 process\_performance(filename)  
 pool\_performance(filename)

**Process** :: I/O :: Total time taken 0.10791778564453125  
**Pool** :: I/O :: Total time taken 0.18415570259094238