Multiprocessing in Python

Clearly Explained!

Multiprocessing

Introduction: Multiprocessing in Python allows you to run multiple processes simultaneously, leveraging multiple CPU cores to speed up your programs. By using the 'multiprocessing' module, you can handle CPU-bound tasks more efficiently, improving your program's performance.

What will you learn in this chapter?

- How multiprocessing works
- Use the 'multiprocessing' module
- Create and manage processes using ProcessPool

Let's explore them today! 🚀

Let's start with an example where we run a simple function twice sequentially (without multiprocessing).

Check this out 👇

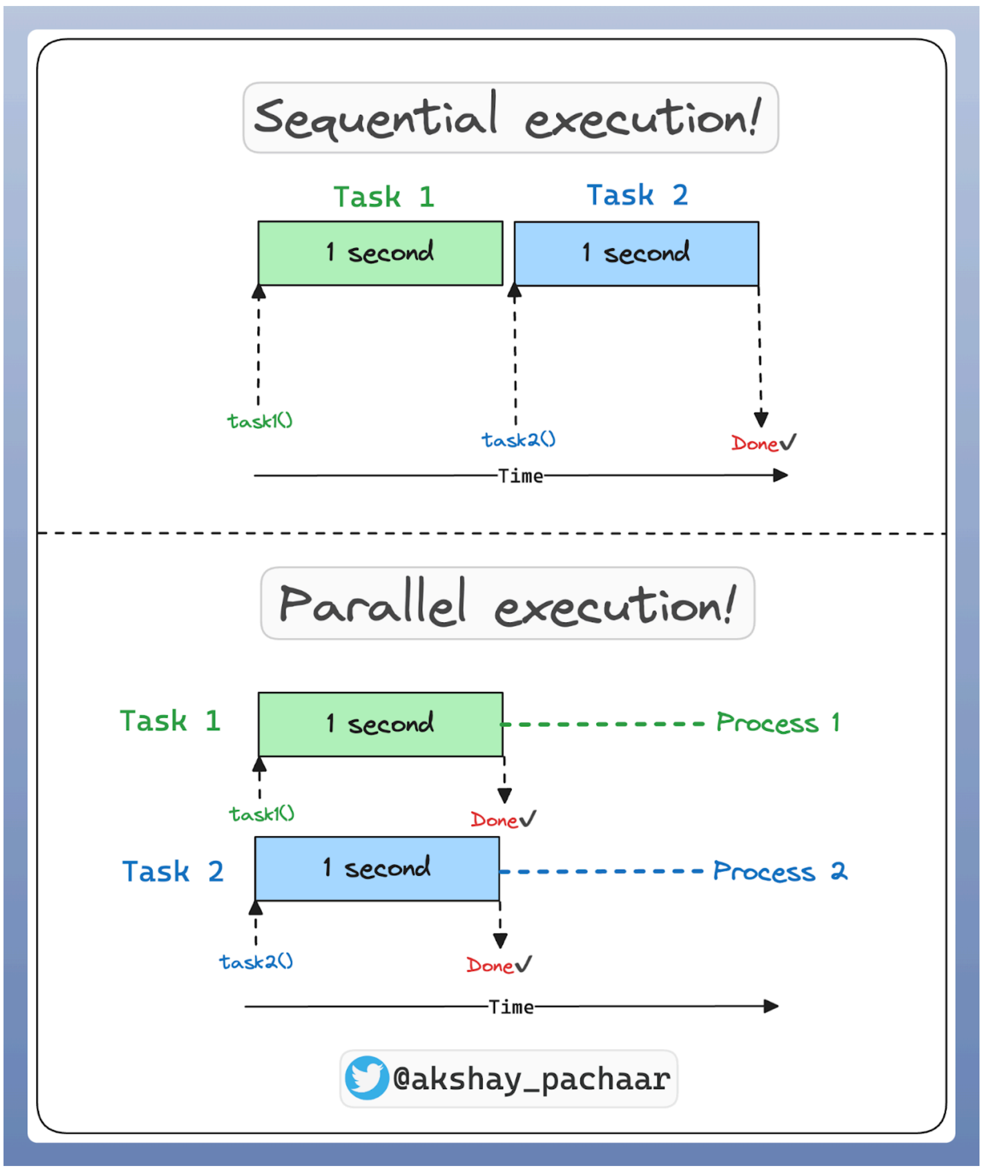
```python
import time

def task(sr_no):
    print(f"Task {sr_no} started")
    time.sleep(1)
    print(f"Task {sr_no} completed")

start = time.time()
task(1) # task 1
task(2) # task 2
end = time.time()

print(f"Execution Time: {end - start:.2f} seconds")
---------------------------------------------------
"""
Task 1 started
Task 1 completed
Task 2 started
Task 2 completed
Execution Time: 2.00 seconds
"""
```

Let's visually understand what happened in the code above & how multi processing can help here.

• Sequential execution: task 2 starts only when task 1 is finished.

• Parallel execution: both tasks are performed at the same time in parallel, on separate CPU cores

Check this 👇



Now that we understand the difference between sequential & parallel execution!

Let's add multiprocessing to the mix and see the difference in execution time! ⏰

Check this out 👇

```python
import multiprocessing

start = time.time()

process1 = multiprocessing.Process(target=task, args=[1])
process2 = multiprocessing.Process(target=task, args=[2])

# start the processes
process1.start()
process2.start()

# join makes sure our processes will finish
# before executing rest of the script
process1.join()
process2.join()

end = time.time()

print(f"Execution Time: {end - start:.2f} seconds")
---------------------------------------------------
"""
Task 1 started
Task 2 started
Task 1 completed
Task 2 completed
Execution Time: 1.02 seconds
"""
```

But why stop there? Let's run our function multiple times using a for loop to see the real power of multiprocessing!

Check this out 👇

```python
processes = []

start = time.time()

# first we start all our processes
for i in range(5):
    process = multiprocessing.Process(target=task, args=[i])
    processes.append(process)
    process.start()

# then we join them
for process in processes:
    process.join()

end = time.time()

print(f"For Loop Execution Time: {end - start:.2f} seconds")
---------------------------------------------------------------------------
"""
Task 0 started
Task 1 started
Task 2 started
Task 3 started
Task 4 started
Task 0 completed
Task 2 completed
Task 1 completed
Task 3 completed
Task 4 completed
For Loop Execution Time: 1.00 seconds
"""
```

To make it even simpler, we can use a ProcessPool!

The recommended way to write multi-processing code in Python.

Check this out 👇

```python
from concurrent.futures import ProcessPoolExecutor, as_completed

start = time.time()

# executor.submit() submits a function to be executed & return a future object.
with ProcessPoolExecutor() as executor:
    # a future object allows us to check the status of submitted function
    futures = [executor.submit(task, i) for i in range(5)]

# as_completed iterates over future objects & return them in order of completion
for f in as_completed(futures):
    f.result()
end = time.time()

print(f"ProcessPool Execution Time: {end - start:.2f} seconds")
-----------------------------------------------------------------------------
"""
Task 0 started
Task 1 started
Task 2 started
Task 3 started
Task 0 completed
Task 4 started
Task 1 completed
Task 2 completed
Task 3 completed
Task 4 completed
ProcessPool Execution Time: 2.04 seconds
"""
```

The tasks start executing in parallel, subject to system resources and process availability!

I ran this code on a system with 4 cpu cores, so tasks 0,1, 2,3 got started & 4 only started after 0 got finished & a core was free!

OK, last but not least let's do one more interesting thing before we wrap it up!

Let's modify task() to take sleep_time as an argument & observe how execution order changes.

Check this out 👇

```python
from concurrent.futures import ProcessPoolExecutor, as_completed

def task_new (sleep_time):

    print(f"Sleeping for {sleep_time} seconds...")
    time.sleep(sleep_time)
    print(f"Done sleeping for {sleep_time} seconds...")

start = time.time()

sleep_times = [3, 2, 1]
# executor.submit() submits a function to be executed & return a future object.
with ProcessPoolExecutor() as executor:
    # a future object allows us to check the status of submitted function
    futures = [executor.submit(task_new, t) for t in sleep_times]

# as_completed iterates over future objects & return them in order of completion
for f in as_completed(futures):
    f.result()
end = time.time()

print(f"ProcessPool Execution Time: {end - start:.2f} seconds")
---------------------------------------------------------------------------
"""
Sleeping for 3 seconds...
Sleeping for 2 seconds...
Sleeping for 1 seconds...
Done sleeping for 1 seconds...
Done sleeping for 2 seconds...
Done sleeping for 3 seconds...
ProcessPool Execution Time: 3.00 seconds
"""
```

Note the order of completion: The task that finishes first is returned first & overall time depends on the lengthiest task.

Multiprocessing is ideal for CPU-bound tasks (intensive calculations, data processing), as each process operates in its own memory space.

Whereas multithreading suits I/O-bound tasks (network requests, file I/O), where threads share memory within the same process.

Enjoyed reading this chapter?

Access the complete book here: **bit.ly/InstantPython**