Demystifying Concurrency in Python

by Amr Abdelkarem

@programmingvalley.com

by Amr AbdElkarem

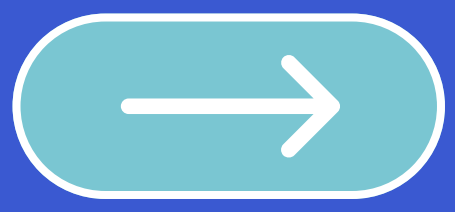
Program, Process & Thread

Program: Static code instructions.

Process: Running instance of a program with isolated memory.

Thread: Executing unit within a process; shares process memory.

@programmingvalley.com



by Amr AbdElkarem

Thread States (Python 3.9)

RUNNING, FINISHED, CANCELLED, NEW, STARTING, WAITING, LOCKED, TERMINATED

Note: “Runnable” is part of WAITING in Python.

by Amr AbdElkarem

Thread States (Python 3.9)

```python
import threading
import time

def worker():
    print(f"Worker thread
{threading.get_ident()} started")
    time.sleep(2)
    print(f"Worker thread
{threading.get_ident()} finished")

t = threading.Thread(target=worker)
print(f"Thread {t.ident} created, state:
{t.is_alive()}")

t.start()
print(f"Thread {t.ident} started, state:
{t.is_alive()}")

time.sleep(1)
print(f"Thread {t.ident} state:
{t.is_alive()}")

t.join()
print(f"Thread {t.ident} joined, state:
{t.is_alive()}")
```

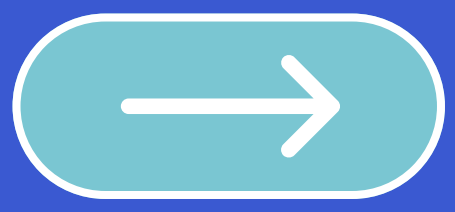
by Amr AbdElkarem

Concurrency vs Parallelism

Concurrency: Multiple tasks making progress via interleaving (context switching).

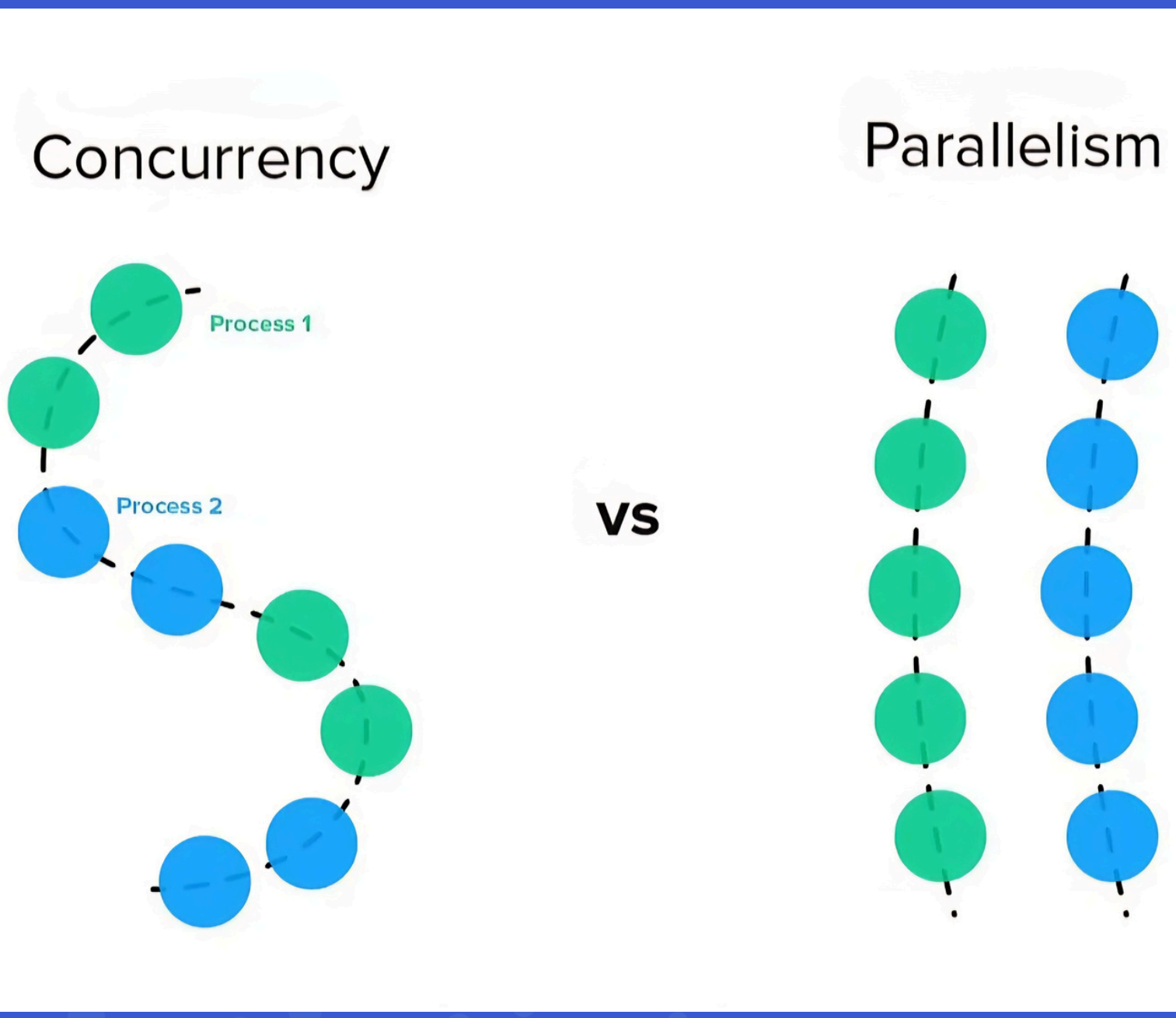
Parallelism: Tasks running at the same time on multiple CPUs/cores.

Concurrency ≠ Parallelism .



by Amr AbdElkarem

Concurrency vs Parallelism



@programmingvalley.com

by Amr AbdElkarem

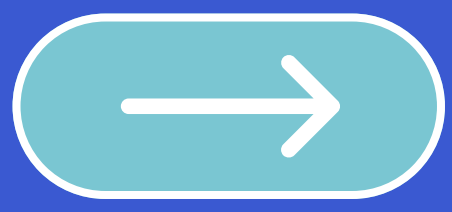
Multithreading in Python

Runs multiple threads in one process.

Threads share memory and are ideal for concurrent I/O.

Example code snippet for running two tasks concurrently.

@programmingvalley.com



by Amr AbdElkarem

Multithreading in Python

```python
import time
from threading import Thread

def task1():
    for i in range(5):
        print("Task 1 is running")
        time.sleep(1)
def task2():
    for i in range(5):
        print("Task 2 is running")
        time.sleep(1)
if __name__ == '__main__':
    # Run tasks concurrently using
multithreading
    t1 = Thread(target=task1)
    t2 = Thread(target=task2)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

@programmingvalley.com

by Amr AbdElkarem

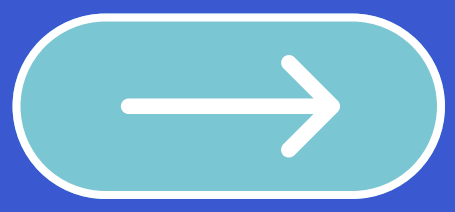
Coroutines & Async/Await

Coroutines: Cooperative multitasking in a single thread.

Use async / await to yield control during blocking I/O.

Great for lightweight concurrent logic with minimal overhead.

@programmingvalley.com

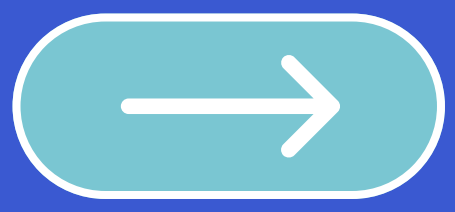


by Amr AbdElkarem

Challenges in Concurrency

Key issues to watch out for:

- Race conditions
- Deadlocks
- Starvation
- Synchronization overhead
- Debugging complexity .



by Amr AbdElkarem

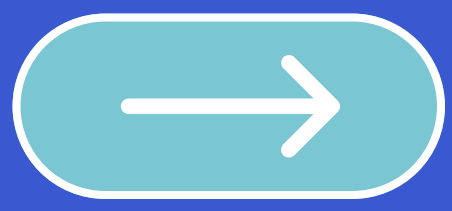
Synchronization Primitives

Mutex: Protect shared resource with lock/unlock.

Semaphore: Control access with counters.

Barrier: Synchronize threads so they wait for each other.
Includes example code for each.

@programmingvalley.com



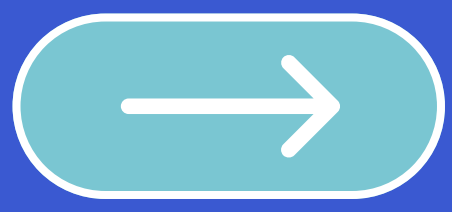
by Amr AbdElkarem

Multiprocessing in Python

Runs multiple processes in parallel, each with isolated memory.

Pros: True parallelism.

Cons: Higher memory and communication overhead, complexity in sharing data.



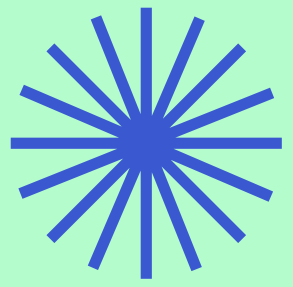
by Amr AbdElkarem

When to use what?

Coroutines — Best for efficient I/O-bound tasks.

Threads — Good for concurrency, less setup, but beware of GIL.

Processes — Use for CPU-intensive tasks needing parallelism.



Find this useful? like and share this post with your friends.

by Amr AbdElkarem
@programmingvalley.com

Save