⚙️ Process vs Thread vs Coroutine: Memory and Scheduling Differences

Understanding the distinctions between **processes**, **threads**, and **coroutines** is crucial in systems programming, performance tuning, and concurrent application design. Each model has unique characteristics in terms of memory isolation, execution context, and scheduling behavior.

🧵 2. Threads

🔍 Definition:

A **thread** is a lightweight unit of execution within a process. Threads share the **same memory space** but have separate stacks and registers.

🧠 Memory:

- Threads within the same process **share heap and global memory**.
- Each thread has its own **stack** and **thread-local storage**.
- Synchronization primitives (mutex, semaphore) are needed to avoid race conditions.

⏱️ Scheduling:

- Also scheduled by the OS.
- Lighter than processes but still requires kernel-level context switching.
- Can run concurrently on multi-core CPUs.

📌 Use Cases:

- Tasks requiring parallel computation or background work
- Web servers (handling multiple client connections)

⚖️ Comparison Table

Feature	Process	Thread	Coroutine
Memory Space	Separate	Shared	Shared (same thread)
Isolation	High	Medium	Low
Scheduling	OS Kernel	OS Kernel	User-space (cooperative)
Context Switch Cost	High	Medium	Low
Creation Overhead	High	Medium	Very Low
Scalability	Low (limited)	Medium	High (thousands possible)
Communication	IPC (complex)	Shared memory	Shared state
Preemption	Yes	Yes	No (manual yield/ await)
Use Case	Isolation, security	Parallel compute	Async I/O, high concurrency

🧪 Python Code Glimpse

```
# Coroutine Example (asyncio)
import asyncio
```

```
async def say_hello():
    await asyncio.sleep(1)
    print("Hello")

async def main():
    await asyncio.gather(say_hello(), say_hello())

asyncio.run(main())
```

```
# Thread Example
from threading import Thread

def say_hello():
    import time
    time.sleep(1)
    print("Hello")

t1 = Thread(target=say_hello)
t2 = Thread(target=say_hello)
t1.start()
t2.start()
t1.join()
t2.join()
```

🧠 Key Takeaways

- **Processes** offer **isolation** but at a higher cost.
- **Threads** enable **true parallelism**, but require synchronization.
- **Coroutines** are ideal for **scalable concurrency** with minimal overhead, especially in **I/O-bound** programs.