🎨Python Concurrency & Parallelism Notes

🧵 Thread

- Smallest unit of execution inside a process.
- Shares memory with other threads in the same process.
- Lightweight, but limited by **GIL** in Python. 👉*Example: A browser tab loading images while still scrolling smoothly.*

Process

- An independent program running in memory.
- Has its own memory, CPU, and resources.
- Can have multiple threads inside it. 👉*Example: Chrome itself is a process, each tab might be another process.*

🔫Spawn

- Starting a **fresh new process** from scratch.
- In Python, `spawn` starts a brand-new interpreter with no inherited state. 👉Clean, safe, but a bit slower.

🥏Fork

- Duplicates the **current process** into a child.
- Child gets a copy of parent's memory/state. 👉Faster, but riskier (may copy unwanted state). *(Default on Unix/Linux)*

Multiprocessing

- Running **multiple processes** at the same time.
- Each process can use a different CPU core. 👉Best for **CPU-bound tasks** (math-heavy, ML training, big data).

Multithreading

- Running **multiple threads** inside one process.
- Great for **I/O-bound tasks** (waiting for files, APIs, network).
- Threads help with **responsiveness** (e.g., GUIs stay smooth while background work runs). 👉Doesn't bypass Python's GIL → not good for CPU-heavy work.

Why use it?

1. **Concurrency (not true parallelism in Python):**
2. While one thread waits (e.g., for network, disk, or input), another keeps working.
3. Makes apps more responsive.
4. **I/O-bound efficiency:**
5. Best for tasks that wait a lot (web requests, file I/O).
6. *Example: A browser loads multiple images at once.*
7. **Responsiveness in GUIs:**
8. Interfaces stay responsive while background threads do heavy work.

👉🏿**Note:** Because of the GIL, only one thread runs Python code at a time. 👉Use **multiprocessing** for CPU-heavy tasks.

🏓Quick Example

```
import threading
import time

def worker(name):
    print(f"Thread {name} starting")
    time.sleep(2)
    print(f"Thread {name} finished")

# Create threads
t1 = threading.Thread(target=worker, args=("A",))
t2 = threading.Thread(target=worker, args=("B",))

# Start threads
t1.start()
t2.start()
```

```
# Wait for threads to finish
t1.join()
t2.join()

print("All threads done!")
```

➡ Both threads run together → total time ≈ 2s, not 4s.

🎏Multithreading Concurrency Example

```
import threading
import time

def download_file(file_id):
    print(f"Downloading file {file_id}...")
    time.sleep(2)  # Simulate download delay
    print(f"File {file_id} downloaded!")

files = [1, 2, 3, 4, 5]
threads = []

# Start threads for multiple downloads
for f in files:
    t = threading.Thread(target=download_file, args=(f,))
    threads.append(t)
    t.start()

# Wait for all threads to finish
for t in threads:
    t.join()

print("All files downloaded!")
```

👉Here, 5 file downloads run concurrently. Instead of waiting 10s (sequential), it finishes in \~2s.

🏓Using ThreadPoolExecutor

```
from concurrent.futures import ThreadPoolExecutor
import time

def download_file(file_id):
    print(f"Downloading file {file_id}...")
    time.sleep(2)
    return f"File {file_id} downloaded!"
```

```
files = [1, 2, 3, 4, 5]

with ThreadPoolExecutor(max_workers=3) as executor:
    results = executor.map(download_file, files)

for result in results:
    print(result)
```

➡ Runs 5 downloads using a pool of 3 threads. Efficient and simpler than managing threads manually.

🎏Parallelism

- True **simultaneous execution**. 👉*Example: 4 CPU cores crunching 4 problems at the same exact time.*

🎙️ Concurrency

- Multiple tasks **taking turns efficiently**.
- They may not run at the *exact same instant*, but they progress together. 👉*Example: Cooking → boil water (wait), chop veggies (work), stir sauce (work).*

🎴 Putting it Together

- **Thread** = worker inside a process.
- **Process** = independent program (can have threads).
- **Spawn** = start new blank process.
- **Fork** = copy current process.
- **Multiprocessing** = many processes in parallel.
- **Multithreading** = many threads in one process.
- **Parallelism** = truly running side by side.
- **Concurrency** = tasks interleaving, feels simultaneous.

🏓Multiprocessing in Python

Why use it?

- Avoids **GIL**.
- Best for CPU-heavy work.

Features

1. **Process class** → create and start processes.

2. **Pool** → manage worker processes.
3. **Queues & Pipes** → exchange data.
4. **Shared memory** → share state safely.

🖌️Example: Creating Processes

```
import multiprocessing
import time

def worker(name):
    print(f"Process {name} starting")
    time.sleep(2)
    print(f"Process {name} finished")

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=worker, args=("A",))
    p2 = multiprocessing.Process(target=worker, args=("B",))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    print("All processes done!")
```

👉Runs in parallel → \~2s instead of \~4s.

🧵 Example: Pool

```
from multiprocessing import Pool
import time

def square(n):
    time.sleep(1)
    return n * n

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]
    with Pool(processes=3) as pool:
        results = pool.map(square, numbers)
```

```
    print(results)
```

➡ 5 tasks handled by 3 workers → finishes faster.

Multiprocessing vs Multithreading

Feature	Multithreading	Multiprocessing
Runs on	Single core (GIL bound)	Multiple cores
Best for	I/O-bound tasks	CPU-bound tasks
Memory	Shared	Separate per process
Overhead	Low	Higher (process start cost)

🎏 Concurrency Features in Python

1. `threading` → threads (I/O-bound).
2. `multiprocessing` → processes (CPU-bound).
3. `asyncio` → coroutines (I/O-heavy, many tasks).
4. `concurrent.futures` → simple wrapper for threads & processes.