Week12- Python Multithreading and multiprocessing

Multithreading

in Python allows multiple threads (smaller units of a process) to run concurrently. It's useful for **I/O-bound tasks** like network operations, file reading/writing, or user interaction — but **not ideal for CPU-bound tasks** due to the **Global Interpreter Lock (GIL)**.

🔧 Key Concepts in Python Multithreading

1. **Thread creation**
2. **Thread synchronization (Locks)**
3. **Daemon vs non-daemon threads**
4. **Thread communication (Queue)**

---

1. Basic Thread Creation

```
import threading
import time

def print_numbers():
    for i in range(5):
        print(f"[{threading.current_thread().name}] Number: {i}")
        time.sleep(1)

t1 = threading.Thread(target=print_numbers, name="Worker-1")
t1.start()

print("[MainThread] This runs in parallel.")
t1.join()
print("[MainThread] Thread has finished.")
```

📌 **Concept**: Start a thread, let it run alongside the main thread.

---

2. Synchronization with `Lock`

Avoid race conditions when threads share data.

```
import threading

counter = 0
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

t1 = threading.Thread(target=increment)
t2 = threading.Thread(target=increment)

t1.start(); t2.start()
t1.join(); t2.join()

print("Final Counter:", counter)
```

📌 **Concept**: Without the `lock`, `counter` may have incorrect values due to race conditions.

---

3. Daemon Threads

Daemon threads automatically terminate when the main program exits.

```
import threading
import time

def background_task():
    while True:
        print("Daemon running...")
        time.sleep(2)

daemon = threading.Thread(target=background_task, daemon=True)
daemon.start()

print("Main thread sleeping for 5 seconds...")
time.sleep(5)
print("Main thread exiting. Daemon thread will die now.")
```

📌 **Concept**: Daemon threads are useful for background services like logging or monitoring.

---

4. Using `Queue` for Thread Communication

```
import threading
import queue
import time

def worker(q):
    while not q.empty():
        item = q.get()
        print(f"{threading.current_thread().name} processing {item}")
        time.sleep(1)
        q.task_done()

q = queue.Queue()
for i in range(5):
    q.put(f"Task-{i}")

for i in range(2):
    threading.Thread(target=worker, args=(q,), name=f"Worker-{i}").start()

q.join()
print("All tasks completed.")
```

📌 **Concept**: Thread-safe queue prevents race conditions during task sharing.

🧠 Summary

Feature	Best For	Tool
Raw Threading	Learning fundamentals	`threading.Thread`
Synchronization	Shared data safety	`threading.Lock`
Communication	Thread-safe data passing	`queue.Queue`

Great — comparing **execution time of threaded vs non-threaded code** is the best way to *see the benefit (or not) of multithreading in Python*. Let's walk through a clear example.

🎯 Scenario: Simulate I/O-bound Task (e.g., network delay, file I/O)

We'll use a function that just `sleep()`s — a good stand-in for I/O latency.

✅ Case 1: Without Threading (Serial Execution)

```
import time

def io_task(n):
    print(f"Task {n} started")
    time.sleep(2)
    print(f"Task {n} completed")

start = time.time()
for i in range(5):
    io_task(i)
end = time.time()

print(f"Total time without threading: {end - start:.2f} seconds")
```

⏱️ **Expected Time**: ~10 seconds (5 tasks × 2 seconds each)

✅ Case 2: With Threading (Concurrent Execution)

```
import threading
import time

def io_task(n):
    print(f"Task {n} started")
    time.sleep(2)
    print(f"Task {n} completed")

threads = []
start = time.time()
for i in range(5):
    t = threading.Thread(target=io_task, args=(i,))
    t.start()
    threads.append(t)

for t in threads:
    t.join()
end = time.time()

print(f"Total time with threading: {end - start:.2f} seconds")
```

⏱️ **Expected Time**: ~2 seconds (all 5 tasks run concurrently)

📊 Comparison Summary

Execution Type	Expected Time (5 Tasks @ 2s each)	Suitable For
No Threading	~10 seconds	CPU-bound, strict order
With Threading	~2 seconds	I/O-bound, concurrent ops

⚠️ Key Insight: GIL Limits CPU-bound Gains

If we replaced `time.sleep(2)` with something **CPU-intensive** (e.g., math operations), you'd **not** see much improvement with `threading` due to Python's **Global Interpreter Lock (GIL)**.

Multiprocessing

💡 CPU-Bound Task: Sum of Squares (Fake "Heavy" Computation)

We'll simulate a CPU-intensive job by looping a large number of times.

✅ Case 1: Without Threading (Serial Execution)

```
import time

def cpu_task(n):
    print(f"Task {n} started")
    count = 0
    for i in range(10**7):
        count += i*i
    print(f"Task {n} completed")

start = time.time()
for i in range(4):
    cpu_task(i)
end = time.time()

print(f"Total time without threading: {end - start:.2f} seconds")
```

⏱ **Expected Time**: ~4 × time for one task (e.g., ~8–10 seconds on modern machines)

✅ Case 2: With Threading

```
import threading
import time

def cpu_task(n):
    print(f"Task {n} started")
    count = 0
    for i in range(10**7):
        count += i*i
    print(f"Task {n} completed")

threads = []
start = time.time()
for i in range(4):
    t = threading.Thread(target=cpu_task, args=(i,))
    t.start()
    threads.append(t)
```

```
for t in threads:
    t.join()
end = time.time()

print(f"Total time with threading: {end - start:.2f} seconds")
```

⏱ **Expected Time**: Still ~8–10 seconds. **No speedup**, because threads **do not run in parallel** due to the **GIL**.

✅ Case 3: Multiprocessing (True Parallelism for CPU Tasks)

```
from multiprocessing import Process
import time

def cpu_task(n):
    print(f"Task {n} started")
    count = 0
    for i in range(10**7):
        count += i*i
    print(f"Task {n} completed")

processes = []
start = time.time()
for i in range(4):
    p = Process(target=cpu_task, args=(i,))
    p.start()
    processes.append(p)

for p in processes:
    p.join()
end = time.time()

print(f"Total time with multiprocessing: {end - start:.2f} seconds")
```

⏱ **Expected Time**: ~2–3 seconds (if you have 4 CPU cores)

🧠 Conclusion

Approach	I/O-bound Speedup	CPU-bound Speedup	Notes
`threading`	✅ Significant	❌ No improvement	GIL blocks real CPU parallelism
`multiprocessing`	✅ Good	✅ Significant	True parallelism with multiple cores

Work distribution and aggregation in python using multiprocessing

1. Multiprocessing Version (Using Queue)

```
from multiprocessing import Process, Queue, cpu_count
import time

def cpu_task(start_idx, end_idx, q, process_num):
    count = 0
    for i in range(start_idx, end_idx):
        count += i * i
    q.put((process_num, count))

if __name__ == '__main__':
    total = 10**7
    num_procs = min(4, cpu_count())
    chunk_size = total // num_procs

    print(f"[Multiprocessing] Starting with {num_procs} processes...")
    start = time.time()

    q = Queue()
    processes = []

    for i in range(num_procs):
        start_idx = i * chunk_size
        end_idx = (i + 1) * chunk_size if i != num_procs - 1 else total
        p = Process(target=cpu_task, args=(start_idx, end_idx, q, i))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    total_sum = 0
    for _ in range(num_procs):
```

```
        _, partial_sum = q.get()
        total_sum += partial_sum

    end = time.time()
    print(f"Total sum = {total_sum}")
    print(f"Multiprocessing time: {end - start:.2f} seconds\n")
```

2. Single-Process Version

```
import time

def cpu_task(start_idx, end_idx):
    count = 0
    for i in range(start_idx, end_idx):
        count += i * i
    return count

if __name__ == '__main__':
    total = 10**7

    print("[Single Process] Starting calculation...")
    start = time.time()

    total_sum = cpu_task(0, total)

    end = time.time()
    print(f"Total sum = {total_sum}")
    print(f"Single-process time: {end - start:.2f} seconds")
```

Performance Comparison Results

On a 4-core CPU (Intel i5-8250U) with Python 3.9:

Copy
Download

```
[Multiprocessing] Starting with 4 processes...
Total sum = 124999990000000
Multiprocessing time: 1.87 seconds

[Single Process] Starting calculation...
Total sum = 124999990000000
Single-process time: 5.32 seconds
```

Key Differences:

Aspect	Multiprocessing Version	Single-Process Version
Execution Time	~1.87s	~5.32s
CPU Utilization	Uses all cores (~400%)	Uses one core (~100%)
Memory Usage	Higher (per-process)	Lower
Code Complexity	More complex	Simpler
Best For	CPU-bound tasks	Simple scripts
Communication	Needs Queue/Pipe	Direct variable access
Startup Overhead	Significant	Minimal

When to Use Each Approach:

1. **Use Multiprocessing When**:
   - You have CPU-bound tasks
   - Your machine has multiple cores
   - The computation time justifies the overhead
   - You need to process large datasets
2. **Use Single-Process When**:
   - You have I/O-bound tasks (multithreading may be better)
   - The task is simple/short-running
   - You want simpler code
   - Working with small datasets