Complete Guide to Python Concurrency

Asyncio, Threading, and Multiprocessing

Comprehensive Learning Material

September 25, 2025

Contents

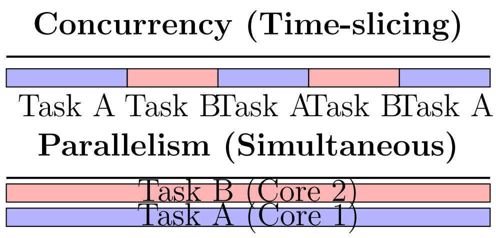
1 Introduction to Concurrency

Key Concept

Concurrency is the ability to handle multiple tasks seemingly at the same time. In Python, we have three main approaches:

- **Asyncio**: Cooperative concurrency (single-threaded)
- **Threading**: Preemptive concurrency (multi-threaded)
- **Multiprocessing**: True parallelism (multi-process)

1.1 Concurrency vs Parallelism



2 Asyncio - Cooperative Concurrency

2.1 Basic Concepts

Key Concept

Asyncio uses an event loop to manage coroutines. Key concepts:

- **Coroutines**: Functions defined with `async def`
- **Await**: Pauses execution and yields control
- **Event Loop**: Manages and executes coroutines
- **Tasks**: Scheduled coroutines

2.2 Simple Asyncio Example

Code Example

```
import asyncio

async def brew_chai():
    print("Brewing chai...")
    await asyncio.sleep(2)  # Non-blocking sleep
    print("Chai is ready")

# Run the coroutine
asyncio.run(brew_chai())
```

What this code does:

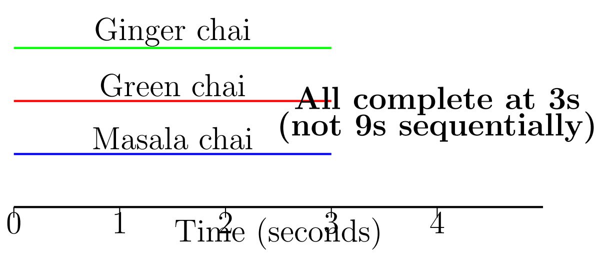
1. Defines an async function (coroutine)
2. Uses `await asyncio.sleep(2)` instead of blocking `time.sleep()`
3. `asyncio.run()` creates event loop and runs the coroutine
4. During the 2-second wait, control can be yielded to other tasks

2.3 Concurrent Execution with asyncio.gather()

Code Example

```
import asyncio

async def brew(name):
    print(f"Brewing {name}...")
    await asyncio.sleep(3)
    print(f"{name} is ready...")

async def main():
    # Run multiple coroutines concurrently
    await asyncio.gather(
        brew("Masala chai"),
        brew("Green chai"),
        brew("Ginger chai"),
    )

asyncio.run(main())
```

Execution Flow:



2.4 HTTP Requests with aiohttp

Code Example

```
import asyncio
import aiohttp

async def fetch_url(session, url):
    async with session.get(url) as response:
        print(f"Fetched {url} with status {response.status}")

async def main():
    urls = ["https://httpbin.org/delay/2"] * 3
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_url(session, url) for url in urls]
        await asyncio.gather(*tasks)

asyncio.run(main())
```

Key Points:

- `aiohttp` provides async HTTP client
- `async with` ensures proper resource cleanup
- All 3 requests run concurrently, completing in 2 seconds total
- `*tasks` unpacks the list for `gather()`

2.5 When to Use Asyncio

Best For	Advantages	Limitations
I/O bound tasks	Memory efficient	Single-threaded
Network requests	No GIL issues	Requires async libraries
File operations	Easy to reason about	CPU-bound tasks blocked
Database queries	Built-in event loop	Learning curve

Table 1: Asyncio Use Cases

3 Threading - Preemptive Concurrency

3.1 Basic Threading Concepts

Key Concept

Threading allows multiple threads to run within a single process:

- **Thread**: Lightweight execution unit
- **GIL**: Global Interpreter Lock (limits true parallelism)
- **Daemon threads**: Die when main program exits
- **Race conditions**: Concurrent access to shared data

3.2 Thread Executors with Asyncio

Code Example

```python
import asyncio
import time
from concurrent.futures import ThreadPoolExecutor

def check_stock(item):
    print(f"Checking {item} in store...")
    time.sleep(3)  # Blocking operation
    return f"{item} stock: 42"

async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        # Run blocking function in thread pool
        result = await loop.run_in_executor(pool, check_stock, "
            Masala chai")
        print(result)

asyncio.run(main())
```

What happens here:

1. `check_stock()` is a blocking function (uses `time.sleep()`)
2. `ThreadPoolExecutor` manages a pool of worker threads
3. `run_in_executor()` runs the function in a separate thread
4. Main asyncio event loop remains responsive

3.3 Daemon vs Non-Daemon Threads

Code Example

```
import threading
import time

def monitor_tea_temp():
    while True:
        print("Monitoring tea temperature...")
        time.sleep(2)

# Daemon thread - dies with main program
t_daemon = threading.Thread(target=monitor_tea_temp, daemon=True)
t_daemon.start()

# Non-daemon thread - keeps program alive
t_normal = threading.Thread(target=monitor_tea_temp)
t_normal.start()

print("Main program done")
```

Thread Type	Daemon=True	Daemon=False
Behavior	Dies when main exits	Keeps program running
Use case	Background monitoring	Critical operations
Example	Logging, heartbeats	Data processing

Table 2: Daemon vs Non-Daemon Threads

3.4 Race Conditions and Thread Safety

Code Example

```
import threading

chai_stock = 0

def restock():
    global chai_stock
    for _ in range(100000):
        chai_stock += 1  # This is NOT atomic!

threads = [threading.Thread(target=restock) for _ in range(2)]

for t in threads:
    t.start()
for t in threads:
    t.join()

print("Chai stock:", chai_stock)  # Often < 200000!
```

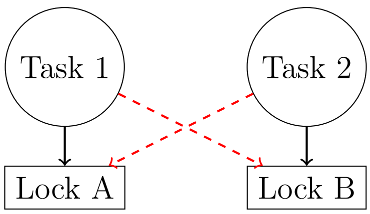
Why the result is wrong:

1. `chai_stock += 1` involves: read → increment → write
2. Two threads can read the same value simultaneously
3. Both increment and write back, losing one increment
4. This is a **race condition**

3.5 Deadlock Example

Code Example

```python
import threading

lock_a = threading.Lock()
lock_b = threading.Lock()

def task1():
    with lock_a:
        print("Task 1 acquired lock a")
        with lock_b:  # Waits for lock_b
            print("Task 1 acquired lock b")

def task2():
    with lock_b:
        print("Task 2 acquired lock b")
        with lock_a:  # Waits for lock_a
            print("Task 2 acquired lock a")

t1 = threading.Thread(target=task1)
t2 = threading.Thread(target=task2)

t1.start()
t2.start()  # This will deadlock!
```



DEADLOCK!
Each waits for other's lock

4 Multiprocessing - True Parallelism

4.1 Process Executors

Code Example

```python
import asyncio
from concurrent.futures import ProcessPoolExecutor

def encrypt(data):
    # CPU-intensive operation
    return f"        {data[::-1]}"

async def main():
    loop = asyncio.get_running_loop()
    with ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(pool, encrypt, "credit_card_1234")
        print(result)

if __name__ == "__main__":
    asyncio.run(main())
```

Key differences from threading:

- Separate Python interpreter per process
- No GIL limitations
- Higher memory overhead
- Inter-process communication needed for shared data

4.2 Mixed Asyncio and Threading

Code Example

```
import asyncio
import threading
import time

def background_worker():
    while True:
        time.sleep(1)
        print("Logging system health          ")

async def fetch_orders():
    await asyncio.sleep(3)
    print("        Order fetched")

# Start background thread
threading.Thread(target=background_worker, daemon=True).start()

# Run async main function
asyncio.run(fetch_orders())
```

This pattern combines:

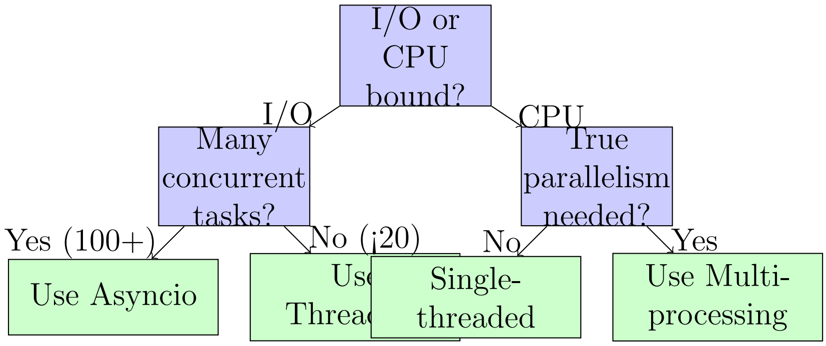
1. Background monitoring thread (daemon)
2. Asyncio event loop for I/O operations
3. Non-blocking cooperation between both

5 Comparison and Decision Guide

Aspect	Asyncio	Threading	Multiprocessing
Best for	I/O bound tasks	Mixed I/O + some CPU	CPU bound tasks
Memory	Very efficient	Moderate overhead	High overhead
GIL Impact	No impact	Limited by GIL	No GIL
Communication	Shared memory	Shared memory + locks	IPC mechanisms
Debugging	Moderate difficulty	Hard (race conditions)	Harder (separate processes)
Scalability	Thousands of tasks	Dozens of threads	Number of CPU cores
Libraries	Requires async libs	Standard libraries work	Standard libraries work

Table 3: Concurrency Method Comparison

6 Decision Flowchart



7 Best Practices

7.1 Asyncio Best Practices

- Always use `async with` for resource management
- Don't mix blocking and async code
- Use `asyncio.gather()` for concurrent tasks
- Handle exceptions in coroutines properly
- Use connection pooling for HTTP clients

7.2 Threading Best Practices

- Use `ThreadPoolExecutor` instead of raw threads
- Always use locks for shared mutable data
- Prefer daemon threads for background tasks
- Avoid circular lock dependencies
- Use `queue.Queue` for thread communication

7.3 General Guidelines

- Start simple - single-threaded first
- Profile before optimizing
- Consider the GIL for CPU-bound tasks
- Use appropriate data structures (thread-safe)
- Plan error handling and resource cleanup

8 Performance Analysis

8.1 Execution Time Comparison

Task Type	Sequential	Asyncio	Threading	Multiprocessing
3 HTTP requests (2s each)	6s	2s	2s	2s
File I/O operations	10s	3s	4s	3s
CPU-intensive calculation	8s	8s	8s*	2s
Mixed I/O + CPU	15s	10s	12s	6s

Table 4: Performance Comparison (*GIL limited)

8.2 Resource Usage

9 Conclusion

Understanding Python's concurrency models is crucial for building efficient applications:

- **Asyncio**: Perfect for I/O-heavy applications with many concurrent operations
- **Threading**: Good for mixed workloads with occasional blocking operations
- **Multiprocessing**: Essential for CPU-bound tasks requiring true parallelism

The key is matching the right tool to your specific use case. Start with profiling your application to understand whether you're I/O or CPU bound, then choose the appropriate concurrency model.

Remember: "Premature optimization is the root of all evil" - start simple and optimize when you have real performance requirements and measurements.