# Python Threading and Concurrency

## 1. Project Overview

This project demonstrates the use of Python’s \*\*threading and concurrency tools\*\* with practical examples. It covers the following core concepts:

- Threading module and Global Interpreter Lock (GIL)

- Locks for synchronization

- Queues for thread-safe communication

- ThreadPoolExecutor for efficient thread management

Two main tasks were implemented:

1. \*\*Image Downloader\*\* – Compare sequential vs. threaded execution.

2. \*\*Race Condition Demo\*\* – Show race conditions and fix them using `Lock`.

The project includes both the \*\*source code\*\* and a \*\*timing performance comparison\*\* table.

---

## 2. Objectives

- To understand the \*\*limitations of Python threading\*\* due to the Global Interpreter Lock (GIL).

- To explore how \*\*Locks\*\* prevent race conditions.

- To implement a practical \*\*threaded image downloader\*\*.

- To demonstrate and fix \*\*race conditions\*\* in file writing.

- To measure performance differences between single-threaded and multi-threaded execution.

---

## 3. Topics Covered

### 3.1 Threading Module & GIL

- The `threading` module allows multiple threads to run within a process.

- The \*\*Global Interpreter Lock (GIL)\*\* ensures that only one thread executes Python bytecode at a time.

- Best suited for \*\*I/O-bound tasks\*\* (e.g., network requests, file I/O).

- Poor choice for \*\*CPU-bound tasks\*\* (e.g., heavy computation).

### 3.2 Locks

- A \*\*Lock\*\* ensures only one thread can access a shared resource at a time.

- Prevents \*\*race conditions\*\* where multiple threads write to the same resource concurrently.

### 3.3 Queue

- `queue.Queue` provides \*\*thread-safe data sharing\*\*.

- Simplifies producer-consumer scenarios.

### 3.4 ThreadPoolExecutor

- Simplifies managing pools of worker threads.

- Abstracts away low-level thread handling.

- Best for managing a large number of parallel I/O operations.

---

## 4. Tasks

### 4.1 Task 1: Image Downloader

- Implemented two versions:

- \*\*Single-threaded\*\*: Downloads 20 images sequentially.

- \*\*Multi-threaded\*\*: Spawns 20 threads to download images concurrently.

- Demonstrates how threading improves performance in I/O-bound tasks.

### 4.2 Task 2: Race Condition Demo

- \*\*Without Lock:\*\* Multiple threads write to the same file → results in corrupted data.

- \*\*With Lock:\*\* Used `threading.Lock` to synchronize file writing → ensures correctness.

---

## 5. Implementation

### 5.1 File Structure

python-threading-concurrency/

│

├── image\_downloader\_single.py # Single-threaded image downloader

├── image\_downloader\_threaded.py # Threaded image downloader

├── race\_condition\_demo.py # Race condition demo without Lock

├── race\_condition\_fixed.py # Fixed race condition with Lock

└── Threading\_and\_Concurrency\_Documentation.md # This documentation

### 5.2 Key Code Highlights

- \*\*Image Downloader (Threaded):\*\*

t = threading.Thread(target=download\_image, args=(url, f"image\_{i}.jpg"))

t.start()

Race Condition Fix:

with lock:

with open(filename, "a") as f:

f.write(text + "\n")

7. Key Learnings

The GIL prevents Python threads from executing CPU-bound tasks in parallel, but threads are still highly effective for I/O-bound operations.

Locks ensure correctness when working with shared resources, at the cost of some performance.

Queues and ThreadPoolExecutor simplify concurrency management and are preferred for production-quality applications.

In practice, threading often improves responsiveness and throughput when tasks involve waiting (e.g., network I/O).