System Requirements Specification Index

For

Python and Memory – Global Interpreter Lock

Version 1.0

IIHT Pvt. Ltd.
fullstack@iiht.com

TABLE OF CONTENTS

1 PROJECT ABSTRACT

NeoBank's backend team needs to improve their transaction processing system. Currently, it struggles to efficiently process multiple transactions concurrently. You've been asked to create a simple demonstration that illustrates how Python's Global Interpreter Lock (GIL) affects concurrent execution and how to work around its limitations. This assignment will help you understand Python's concurrency model and when to use threading versus multiprocessing.

2 BUSINESS REQUIREMENTS:

Screen Name	Console input screen
Problem Statement	1. Demonstrate the impact of GIL on CPU-bound operations 2. Demonstrate the impact of GIL on I/O-bound operations 3. Compare performance between sequential, threading, and multiprocessing approaches 4. Document findings and recommendations for when to use each approach

3 EXPECTED OUTCOMES

3.1 INPUT REQUIREMENTS

1. For CPU-bound tasks:
   - Threading should show minimal to no speedup (due to GIL)
   - Multiprocessing should show significant speedup (approaching the number of cores)

2. For I/O-bound tasks:
    - - Threading should show significant speedup (near the number of threads)
    - - Multiprocessing should also show good speedup but with higher overhead
3. GIL contention demonstration:
    - - Multiple threads incrementing a counter should not be much faster than a single thread
    - - This clearly demonstrates how the GIL prevents true parallelism for CPU-bound operations
4. GIL contention demonstration:
    - - For small tasks, overhead might outweigh performance benefits
    - - When n ≤ 2 in the CPU task, no prime numbers are found (edge case)
    - - Empty task lists should be handled gracefully

3.2 Implementation Constraints

1. CPU-Bound Task:

- - Implement a simple prime number calculator
- - Function should be computationally intensive
- - Example:

```python
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def cpu_intensive_task(n):
    """Calculate sum of primes up to n"""
    return sum(i for i in range(2, n) if is_prime(i))
```

2. I/O-Bound Task:

- - Implement a function that simulates I/O operations
- - Use time.sleep() to simulate waiting for I/O

- Example:
  ```python
  def io_intensive_task(seconds):
      """Simulate an I/O operation that takes 'seconds' to complete"""
      time.sleep(seconds)
      return f"Completed I/O operation that took {seconds} seconds"
  ```

3. Performance Comparison:

- Measure execution time for each approach
- Calculate speedup relative to sequential execution
- Example output:
  ```
  CPU-bound task (10 workers):
  - Sequential: 10.5s
  - Threading: 10.2s (1.03x speedup)
  - Multiprocessing: 2.8s (3.75x speedup)

  I/O-bound task (10 workers):
  - Sequential: 10.0s
  - Threading: 1.05s (9.52x speedup)
  - Multiprocessing: 1.1s (9.09x speedup)
  ```

4. TEMPLATE CODE STRUCTURE:

1. Basic Functions:

- `cpu_intensive_task(n)` - A CPU-bound function that calculates the sum of prime numbers up to n
- `io_intensive_task(seconds)` - An I/O-bound function that simulates waiting for I/O operations

2. Execution Approaches:

- `run_sequential(func, args_list)` - Runs tasks one after another
- `run_threading(func, args_list, num_threads)` - Runs tasks using multiple threads
- `run_multiprocessing(func, args_list, num_processes)` - Runs tasks using multiple processes

3. Analysis Functions:

- `compare_performance(task_func, args_list, num_workers)` - Compares performance of different approaches

4. Main Program Function:

- Demonstrates the impact of GIL on different types of tasks
- Shows when threading is beneficial despite the GIL
- Shows when multiprocessing provides better performance

5. EXECUTION STEPS TO FOLLOW:

1. Run the program with different numbers of workers (1, 2, 4, 8)
2. Observe how the GIL affects threading performance for CPU-bound tasks
3. Observe how threading performs well for I/O-bound tasks despite the GIL
4. Modify the intensity of CPU and I/O tasks to see how it affects performance
5. Document your findings on when to use threading vs. multiprocessing