ASSIGNMENT REPORT :EVALUATING THE OPTIMAL USE OF PROCESSES VERSUS THREADS IN PYTHON PARALLEL PROGRAMMING

CENG 2034, OPERATING SYSTEMS

Barış Berişbek
barisberisbek@posta.mu.edu.tr

Sunday 26th May, 2024

Abstract

This report evaluates the use of processes and threads in Python for parallel programming. It investigates their performance in both I/O-bound and CPU-bound tasks, providing insights into the Global Interpreter Lock (GIL) and its impact on threading in Python. We present practical implementations, performance analysis, and recommendations for optimal usage scenarios.

1 Introduction

In this report, we evaluate the use of processes and threads in Python for parallel programming. We investigate their performance in both I/O-bound and CPU-bound tasks, providing insights into the Global Interpreter Lock (GIL) and its impact on threading in Python.

2 Theoretical Understanding

2.1 Processes and Threads

A process is an independent execution unit that contains its own state information, memory space, and program code. In contrast, a thread is a smaller execution unit that shares the memory space and state information of its parent process.

2.2 Multiprocessing and Threading in Python

Python's `multiprocessing` module allows for the creation of processes, enabling parallel execution of code. The `threading` module, on the other hand, allows for concurrent execution using threads within a single process.

2.3 Global Interpreter Lock (GIL)

The GIL is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes simultaneously. This means that in a multi-threaded Python program, only one thread can execute Python code at a time, which can be a bottleneck for CPU-bound tasks.

2.4 Advantages and Disadvantages of Processes and Threads

- **Processes:**
  - **Advantages:** Independent memory space, no GIL impact, better for CPU-bound tasks.
  - **Disadvantages:** Higher memory consumption, slower inter-process communication.
- **Threads:**
  - **Advantages:** Lower memory consumption, faster inter-thread communication, better for I/O-bound tasks.
  - **Disadvantages:** GIL impact, shared memory space can lead to synchronization issues.

3 Practical Implementation

3.1 I/O-Bound Tasks

We measured the performance of threading and multiprocessing in handling I/O-bound tasks (downloading files).

3.1.1 Threading Implementation

```
import threading
import requests
import time

def download_file(url):
    response = requests.get(url)
    print(f"Downloaded {url} with size {len(response.content)} bytes")

def threaded_io_task():
    urls = [
        'https://file-examples.com/storage/fe15076da466528199d9c5a/2017/10/file_example_JPG_500kB.jpg',
        'https://file-examples.com/storage/fe15076da466528199d9c5a/2017/10/file_example_JPG_1MB.jpg',
        'https://file-examples.com/storage/fe15076da466528199d9c5a/2017/10/file_example_JPG_2500kB.jpg'
    ]

    threads = []
    start_time = time.time()

    for url in urls:
```

```
        thread = threading.Thread(target=download_file, args=(url,))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    end_time = time.time()
    print(f"Threaded I/O-bound task took {end_time - start_time} seconds")

threaded_io_task()
```

Listing 1: Threaded I/O-bound Task

3.1.2 Multiprocessing Implementation

```
import multiprocessing
import requests
import time

def download_file(url):
    response = requests.get(url)
    print(f"Downloaded {url} with size {len(response.content)} bytes")

def process_io_task():
    urls = [
        'https://file-examples.com/storage/fe15076da466528199d9c5a/2017/10/file_example_JPG_500kB.jpg',
        'https://file-examples.com/storage/fe15076da466528199d9c5a/2017/10/file_example_JPG_1MB.jpg',
        'https://file-examples.com/storage/fe15076da466528199d9c5a/2017/10/file_example_JPG_2500kB.jpg'
    ]

    processes = []
    start_time = time.time()

    for url in urls:
        process = multiprocessing.Process(target=download_file, args=(url,))
        processes.append(process)
        process.start()

    for process in processes:
        process.join()

    end_time = time.time()
    print(f"Process I/O-bound task took {end_time - start_time} seconds")

process_io_task()
```

Listing 2: Process I/O-bound Task

3.2 CPU-Bound Tasks

We measured the performance of threading and multiprocessing in handling CPU-bound tasks (computationally intensive calculations).

3.2.1 Threading Implementation

```python
import threading
import time

def cpu_intensive_task(n):
    count = 0
    for i in range(n):
        count += i
    print(f"Completed task with count {count}")

def threaded_cpu_task():
    n = 10**7
    threads = []
    start_time = time.time()

    for _ in range(4):
        thread = threading.Thread(target=cpu_intensive_task, args=(n,))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    end_time = time.time()
    print(f"Threaded CPU-bound task took {end_time - start_time} seconds")

threaded_cpu_task()
```

Listing 3: Threaded CPU-bound Task

3.2.2 Multiprocessing Implementation

```python
import multiprocessing
import time

def cpu_intensive_task(n):
    count = 0
    for i in range(n):
        count += i
    print(f"Completed task with count {count}")

def process_cpu_task():
    n = 10**7
    processes = []
    start_time = time.time()
```

```
14
15      for _ in range(4):
16          process = multiprocessing.Process(target=cpu_intensive_task, args
                =(n,))
17          processes.append(process)
18          process.start()
19
20      for process in processes:
21          process.join()
22
23      end_time = time.time()
24      print(f"Process CPU-bound task took {end_time - start_time} seconds")
25
26  process_cpu_task()
```

Listing 4: Process CPU-bound Task

4 Performance Analysis

We collected performance metrics such as execution time, CPU usage, and memory consumption for both scenarios.

4.1 I/O-Bound Task Performance

Approach	Execution Time (s)	Memory Usage (MB)
Threading	1.95	25.12
Multiprocessing	7.63	28.24

Table 1: I/O-bound task performance comparison

Figure 1: I/O-bound task performance comparison

Figure 2: Barış Berişbek'in bilgisayarından alınmış bir ekran fotoğrafıdır.

Approach	Execution Time (s)	Memory Usage (MB)
Threading	1.63	28.25
Multiprocessing	0.97	28.28

Table 2: CPU-bound task performance comparison

```
Threaded CPU-bound task performance:
Threaded CPU-bound task took 1.6396141052246094 seconds
Execution Time: 1.6396141052246094 seconds
Memory Usage: 28.25 MB

Process CPU-bound task performance:
Process CPU-bound task took 0.97867751121521 seconds
Execution Time: 0.97867751121521 seconds
Memory Usage: 28.2890625 MB

Process finished with exit code 0
```

Figure 3: CPU-bound task performance comparison

Figure 4: Barış Berişbek'in bilgisayarından alınmış bir ekran fotoğrafıdır.

4.2 CPU-Bound Task Performance

5 Conclusion and Recommendations

Based on our performance analysis, we observe the following:

- For I/O-bound tasks, threading significantly outperforms multiprocessing. This is because threading allows for better utilization of waiting times inherent in I/O operations.
- For CPU-bound tasks, threading is faster than multiprocessing due to the overhead associated with process creation and inter-process communication. However, the GIL may limit the effectiveness of threading for more complex CPU-bound tasks.
- The GIL has a significant impact on the performance of multi-threaded Python programs, especially for CPU-bound tasks. Multiprocessing can bypass the GIL, but at the cost of higher memory usage and slower inter-process communication.