ASSIGNMENT REPORT 1: PROCESS AND THREAD IMPLEMENTATION

OS 2034, OPERATING SYSTEMS

Ayselin Aydoğdu
ayselinaydogdu@posta.mu.edu.tr

Monday 21st April, 2025

Abstract

In this project, I worked on performance of two main approaches which are multi_threading and multi_processing. I used both approaches in different type of tasks like CPU-Bound Tasks and I/O-Bound Tasks. The main purposes of this project are realizing which approach works faster in different types of tasks and understanding that in which situations we should use multi_processing or multi_threading. For CPU-Bound Tasks, I found prime numbers between 0 and 4000000 with both multi_processing and multi_threading approach and I noted that the execution times for both approaches. Results showed me that multi_processing works faster in CPU-Bound Tasks compared to multi_threading. Also, I used both approaches in order to downloading multiple files which is known as I/O-Bound Task. Again I measured the execution time for both approaches and I realized that execution time for multi_threading is less than multi_processing. I learned that multi_threading works faster in I/O-Bound Tasks compared to multi_threading.

1 Introduction

Parallel programming is a significant method in order to improve the performance of programs. Processes and threads are two major ways to run code at the same time. There are some important properties for both processes and threads. First of all, threads share same memory however, processes use multiple CPU cores and run separately. I worked on the performance of both processes and threads with different kind of tasks such as CPU-Bound Tasks and I/O-Bound Tasks. I/O-Bound Tasks simply known as tasks that don't use CPU intensively like downloading multiple files while CPU-Bound Tasks work on CPU intensively like prime number calculation. I realized during the project multi_processing works faster on CPU-Bound Tasks and multi_threading works faster on I/O-Bound Tasks. Because, there is a special system in Python called GIL and because of this system only one thread can work at a time. Therefore, multi_threading start to be less efficient way in order to CPU-Bound Tasks compared to multi_processing. It is important to note that, choosing and working on which approaches like multi_threading or multi_processing is crucial topic. Because choosing wrong method can make program slower and it can put the user in a difficult situation. During this project, I learned both pros and cons for multi_threading and multi_processing. In this way, I can generalize which approach I should use in different scenarios.

2 Assignments

In this section of the project, I wrote different Python codes using multi_processing and multi_threading to handle different tasks, such as I/O-bound tasks and CPU-bound tasks. In each subsection, I included explanations about the tasks, multi_processing, and multi_threading.

2.1 Scenario 1: I/O-Bound Tasks

In this section, I implemented multi_threading to handle with I/O-Bound Task which is downloading multiple files. The time module that is located into this code was used to measure the start and finish time of the task. As a result of this code, parallel downloading shows better performance compared to downloading files one by one.

```
import multiprocessing
import requests
import time
import os
def downloading(url, name):
    try:
        print("The file " , name, " is downloading.")
        response = requests.get(url)
        with open(name, 'wb') as f:
            f.write(response.content)

        print("The file " , name, " is downloaded.")
    except Exception as hata:
        print(f"{name} hatası: {str(hata)}")

if __name__ == '__main__':
    dosya_list = [
        ("https://aad216.a-cdn.akinoncloud.com/products/2024/10/07/11917890/2b454b9b-c291-4de6-9a
        ("https://narasport.com/image/cache/catalog/Helix/Icon/Voleybol%20Icon%202-1500x1500.jpg"
        ("https://minio.yalispor.com.tr/sneakscloud/blog/basketbol-topu-ozellikleri-01_6053563b5b
    ]
    processes = []
    start = time.time()

    for url, name in dosya_list:
        process = multiprocessing.Process(target=downloading, args=(url, name))
        processes.append((process, name))
        process.start()

    for process, name in processes:
        process.join()
    finish = time.time()
    total_time = finish - start
    print(f"Total time: {total_time:.2f} seconds")
```

2.2 Scenario 1: I/O-Bound Tasks

In this section, I used multi_processing to download several files at the same time. For each downloading step, a new process was created. Unlike threading, where all threads share the same memory, each process in multi_processing runs separately. I used the time module to measure the total time between starting and finishing. As a result of this code, multi_processing made the downloads faster compared to downloading files one by one.

```
import multiprocessing
import requests
import time
import os
def downloading(url, name):
    try:
        print("The file " , name, " is downloading.")
        response = requests.get(url)
        with open(name, 'wb') as f:
            f.write(response.content)

        print("The file " , name, " is downloaded.")
    except Exception as hata:
        print(f"{name} hatası: {str(hata)}")

if __name__ == '__main__':
    dosya_list = [
        ("https://aad216.a-cdn.akinoncloud.com/products/2024/10/07/11917890/2b454b9b-c291-4de6-9a
        ("https://narasport.com/image/cache/catalog/Helix/Icon/Voleybol%20Icon%202-1500x1500.jpg"
        ("https://minio.yalispor.com.tr/sneakscloud/blog/basketbol-topu-ozellikleri-01_6053563b5b
    ]
    processes = []
    start = time.time()

    for url, name in dosya_list:
        process = multiprocessing.Process(target=downloading, args=(url, name))
        processes.append((process, name))
        process.start()

    for process, name in processes:
        process.join()
    finish = time.time()
    total_time = finish - start
    print(f"Total time: {total_time:.2f} seconds")
```

2.3 Compare

In this project, I used both multi_threading and multi_processing in order to download files. With multi_threading, one thread waits for file to download while other threads can continue their working. In multi_processing, each downloading steps run in a separate processes. It is known that downloading files is an I/O-Bound Task. Although both methods work at almost the same speed, multi_threading is most useful way compared to multi_processing in order to work on I/O-Bound Tasks.

2.4 Scenario 2: CPU-Bound Tasks

In this section, I implemented multi_threading approach in order to find prime numbers between 0 and 4000000. I used multiple threads to process different sub-ranges of the numbers at the same time. As a result, execution time can be shorter than single thread approach.

```
import multiprocessing
import time
import math
def PrimeOrNot(start, end, prime_number_list):
    for number in range(start, end+1):
        if number < 2:
            continue
        x = True
        for i in range(2, int(math.sqrt(number))+1):
            if number % i == 0:
                x = False
                break
        if x :
            prime_number_list.append(number)
if __name__ == '__main__':
    start = 0
    end = 100
    process_number = 4
    ranges = [(0, 1000000), (1000000, 2000000), (2000000, 3000000), (3000000, 4000000)]
    manager = multiprocessing.Manager()
    prime_number_list = manager.list()
    processes = []
    start_time = time.time()

    for start, end in ranges:
        p = multiprocessing.Process(target= PrimeOrNot, args= (start, end, prime_number_list))
        processes.append(p)
        p.start()
    for p in processes:
        p.join()
    finish_time = time.time()
    total_time = finish_time - start_time
#    print("Prime Numbers: ",  sorted(prime_number_list))
    print("Total number of prime numbers between 0 and 4000000: ",  len(prime_number_list))
    print(f"Total Time: {total_time} seconds")
```

2.5 Scenario 2: CPU-Bound Tasks

In this section, the CPU-Bound Task of finding prime numbers was executed by multi_processing. The code divides range into four different parts and each process works in parallel in parts. Each process finds the prime numbers between its specified range and appends numbers to the shared list which all processes append their numbers to list. I used time module in order to measure total execution time and as a result, using multi_processing reduced the total execution time.

```
import multiprocessing
import time
import math
def PrimeOrNot(start, end, prime_number_list):
    for number in range(start, end+1):
        if number < 2:
            continue
        x = True
        for i in range(2, int(math.sqrt(number))+1):
            if number % i == 0:
                x = False
                break
        if x :
            prime_number_list.append(number)
if __name__ == '__main__':
    start = 0
    end = 100
    process_number = 4
    ranges = [(0, 1000000), (1000000, 2000000), (2000000, 3000000), (3000000, 4000000)]
    manager = multiprocessing.Manager()
    prime_number_list = manager.list()
    processes = []
    start_time = time.time()

    for start, end in ranges:
        p = multiprocessing.Process(target= PrimeOrNot, args= (start, end, prime_number_list))
        processes.append(p)
        p.start()
    for p in processes:
        p.join()

    finish_time = time.time()
    total_time = finish_time - start_time
#    print("Prime Numbers: ",  sorted(prime_number_list))
    print("Total number of prime numbers between 0 and 4000000: ",  len(prime_number_list))
    print(f"Total Time: {total_time} seconds")
```

2.6 Compare

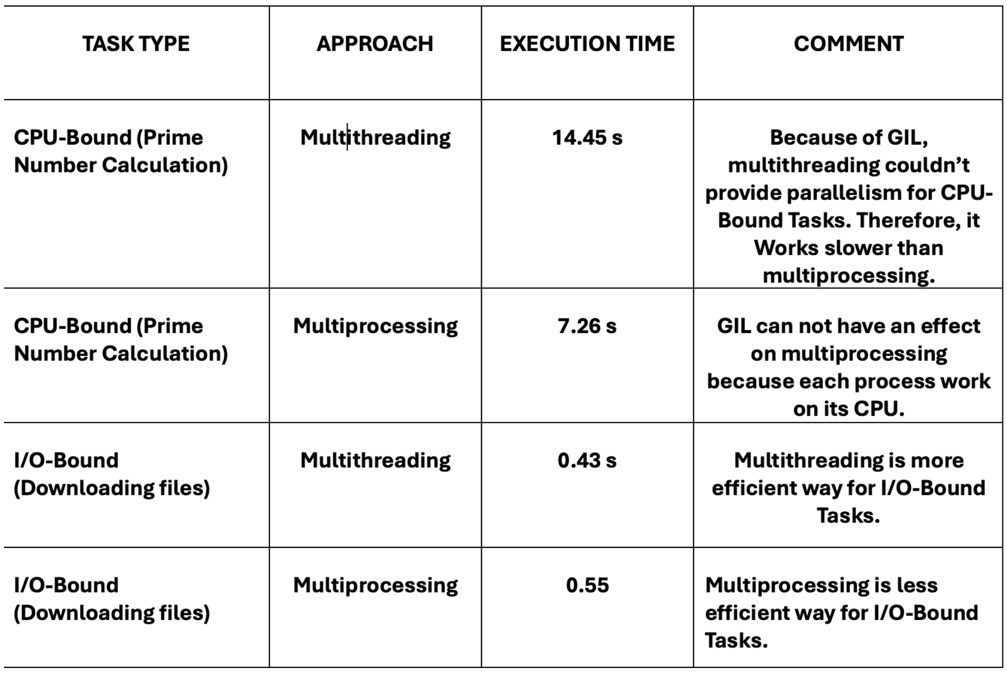
In this project, I used both multi_processing and multi_threading in order to find prime numbers which is known as CPU-Bound Task. In multi_processing approach, each process runs on a different CPU core and they can work at the same time. This situation makes the program more faster. In multi_threading approach; due to GIL in Python, CPU-intensive jobs like prime number calculation cannot have multiple threads at the same time. This situation makes difficult to improve performance. Therefore, multi_processing gives user more advantage in terms of speed and performance compared to multi_processing. On the other hand, there is a special system in Python that is called GIL. Because of this system, only one thread can work at the same time. In small ranges like finding prime numbers between 0 and 1000, GIL doesn't make any problem and multi_threading can work faster in small ranges compared to multi_processing because I tried and experienced this situation. However; between huge ranges like 0 and 4000000, each thread work in very long time therefore threading started to be less efficient way for finding prime numbers between huge ranges. If we use huge ranges in order to find prime numbers, we should use multi_processing approach.

3 Results

In this project, I compared the performance of multi_processing and multi_threading for CPU-Bound Tasks and I/O-Bound Tasks. First of all, I use both multi_processing and multi_threading approach for CPU-Bound Task which is finding prime numbers between 0 and 4000000. I measured execution time in both approaches and I realize that multi_processing offers less execution time therefore, it offers better way while finding prime numbers which is very CPU-intensive job. I learned that multi_processing is more useful way in order to work on CPU-Bound Tasks compared to multi_threading. Because of GIL in Python, only one thread can work at a time therefore, this situation makes difficult to improve performance. Secondly, I use again both approaches multi_processing and multi_threading for I/O-Bound Task which is downloading multiple files. I measured the execution time in both approaches and I realize that execution time in multi_threading approach is less compared to multi_processing. Because we know that downloading files is an I/O-Bound Task and multi_threading works better in I/O-Bound Tasks.

4 Conclusion

In this project, I made performance analysis about multi_threading and multi_processing on different tasks such as CPU-Bound Tasks and I/O-Bound Tasks. From my point of view, multi_processing works better and in a efficient way on CPU-Bound Tasks like finding prime numbers between two ranges. Because each process can run at the same time on a different CPU cores. Therefore, they can not blocked by GIL. However; if we use multi_threading in same prime number calculation example, only one thread can run at a time due to GIL in Python. GIL allows only one thread can run at a time on CPU and other threads must wait. This situation makes difficult to improve the performance. On the other hand, multi_threading works faster on I/O-Bound Tasks like downloading multiple files at the same time. GIL does not create any big problem, because most of the time in I/O-Bound Tasks was spent waiting not using the CPU. We can generalize the situations easily: For example; if any task which uses CPU intensively, we should use the multi_processing approach. On the other hand, if any task which doesn't use CPU intensively and it spends its time especially in waiting, we should use multi_threading.



TASK TYPE	APPROACH	EXECUTION TIME	COMMENT
CPU-Bound (Prime Number Calculation)	Multithreading	14.45 s	Because of GIL, multithreading couldn't provide parallelism for CPU-Bound Tasks. Therefore, it Works slower than multiprocessing.
CPU-Bound (Prime Number Calculation)	Multiprocessing	7.26 s	GIL can not have an effect on multiprocessing because each process work on its CPU.
I/O-Bound (Downloading files)	Multithreading	0.43 s	Multithreading is more efficient way for I/O-Bound Tasks.
I/O-Bound (Downloading files)	Multiprocessing	0.55	Multiprocessing is less efficient way for I/O-Bound Tasks.

Figure 1: CPU-Bound ve I/O-Bound görevler için Multi_threading ve Multi_processing karşılaştırması.

References

[1] Python Software Foundation. multiprocessing — process-based parallelism.

[2] Python Software Foundation. threading — thread-based parallelism.

[3] GeeksforGeeks. Multithreading and multiprocessing in python.

[4] Stack Overflow. Python concurrency: Multithreading vs multiprocessing discussions.

[5] Real Python. Concurrency in python: Threading, multiprocessing, and asyncio.

[6] Wikipedia contributors. Global interpreter lock.