Department of
Computer Science and Engineering
Independent University, Bangladesh

Assignment No : Assignment 1
Assignment Topics : Multiprocessing and Multithreading

Serial No.	**01**
ID	**2220281**
Name	**Mst. Aysa Siddika Meem**
Section	**01**

Course Code	**CSC470**
Course Title	**Introduction to Parallel Programming**
Instructor	**Dr. Md. Rashedur Rahman**
Submission Date	**20/11/2025**

Total Marks	**100 Marks**
Obtained Marks	
Comments	
Signature	

Task 1: Understanding Parallelism

a) The difference between multiprocessing and multithreading.

Answer:

Multiprocessing: A system with more than one or two processors is known as multiprocessing. CPUs are added in multiprocessing to speed up system computation. Multiprocessing allows for the execution of numerous processes at once. Study up on related subjects. Two types of multiprocessing are distinguished:
1. Equitable Multiprocessing
2. Inconsistent Multiprocessing

Multithreading : Multithreading is a system in which several threads are formed from a single process to increase the system's computational speed. Multithreading involves the execution of several threads of a process at the same time, and process creation is done in an inexpensive manner.

Difference between multiprocessing and multithreading:

SL	Title	Multiprocessing	Multithreading
1	Computing Power	In Multiprocessing, CPUs are added for increasing computing power.	Multithreading increases computational capacity by creating several threads from a single process.
2	Classification	Classified into Symmetric and Asymmetric multiprocessing.	No categories exist for multithreading.
3	Creation Time	Process creation takes more time and overhead.	Thread creation is faster and more economical.
4	Memory Model	Each process has its own memory space (separate houses).	Threads share the same memory: all threads inside one process use the same address space, like roommates sharing one apartment.
5	Weight	Processes are heavyweight and need more resources.	Threads are lightweight and use fewer resources.
6	Execution	Many processes run simultaneously across multiple CPU cores.	Many threads within one process run simultaneously.
7	Communication	Safer communication using queues/pipes, reducing interference.	Easier communication using shared variables but riskier due to shared memory.
8	Crash Impact	Crash is isolated: one process crash doesn’t affect others.	One thread crash can crash the entire program.

9	GIL Limitation	No GIL issue: each process can run independently on CPU cores.	GIL limits Python threads for CPU-heavy tasks.
10	Best Use Case	Suitable for CPU-bound tasks (ML, scientific computation, image processing).	Suitable for I/O-bound tasks (file I/O, web requests, networking).
11	Context Switching	Slower context switching due to separate memory spaces.	Faster context switching because threads share memory.
12	Memory Usage	Uses more memory per process.	Uses less memory because threads share data.
13	Analogy	Works like workers in separate rooms: safe but slower communication.	Works like workers at one table fast but more interference.
14	Race Conditions	Very low chance of race conditions since processes don't share memory.	Higher chance of race conditions; locks needed.
15	Overhead	High overhead due to creating and managing multiple processes.	Low overhead because threads are cheaper and easier to manage.

b) When to use process-based parallelism vs. thread-based parallelism in Python.

Process-based parallelism is most appropriate for CPU-intensive workloads, where the program performs substantial computational tasks such as numerical calculations, data analysis, machine-learning model training, encryption, or video processing. These operations place a high demand on the processor, and multiprocessing is suitable because each process runs independently on a separate CPU core. This design bypasses Python's Global Interpreter Lock (GIL) and enables true parallel execution, which can significantly improve performance in computation-heavy scenarios.

In contrast, thread-based parallelism is better suited for I/O-bound tasks that spend a large portion of their execution time waiting for external operations to complete. Examples include handling network requests, reading or writing files, interacting with APIs, or waiting for user input. Since these tasks do not continuously occupy the CPU, multithreading allows another thread to run while one is waiting, thereby improving overall responsiveness and efficiency without being limited by the GIL.

Finally we can say that:

- Use multiprocessing for CPU-bound tasks that require intensive computation.
- Use multithreading for I/O-bound tasks where most time is spent waiting rather than computing.

Task 2: Implementing a CPU-Bound Task Using Multiprocessing (30 Marks) Write a Python program that computes the sum of squares for a large range of numbers.
a) Implement both sequential and parallel versions using multiprocessing.
b) Measure the execution time for both versions using time.time().
c) Compare the performance and write a short conclusion.
Hints:
• Use Pool from multiprocessing to create worker processes.
• Try calculating the sum of squares for range(1, 107).**

Answer:

(a) Sequential and Parallel Implementation

I wrote two versions of the sum-of-squares program: a sequential version that runs on a single process, and a parallel version that uses `multiprocessing.Pool` to split the range into chunks and compute them across multiple processes.

(b) Execution Time Measurement

I used `time.time()` to record the start and end time of both versions. This allowed me to compare how long the sequential and multiprocessing implementations took to finish the same task.

Link: Assignment_1-2220281-Mst. Aysa Siddika Meem.ipynb

```
...
    Computing sum of squares from 1 to 10000000
    Running Sequential Version...
    Sequential Result   : 333333383333335000000
    Sequential Time     : 0.8235 seconds

    Running Parallel Version using 2 processes...
    Parallel Result     : 333333383333335000000
    Parallel Time       : 0.8911 seconds

    Both versions produced the same result.
```

c) Performance comparison and conclusion

For this experiment, the program computed the sum of squares from 1 to 10^7 using both a sequential implementation and a multiprocessing implementation with 2 processes. The sequential version produced the correct result in approximately 0.8235 seconds, whereas the parallel version using a process pool required about 0.8911 seconds to compute the same result. Although multiprocessing is theoretically more suitable for CPU-bound tasks, in this case the parallel version was slightly slower. The main reason is the overhead of multiprocessing: creating processes, dividing the range into chunks, and collecting and combining partial results all introduce extra cost. For this particular input size and only two processes, that overhead outweighs the benefit of parallel execution. In summary, the results show that multiprocessing does not always guarantee a speedup; it is most effective for

larger workloads or when using more CPU cores, where the computation time dominates the process-management overhead.

Task 3: Implementing an I/O-Bound Task Using Multithreading (30 Marks) Simulate an I/O-bound task by downloading multiple web pages using Python's threading module.
a) Implement a function that fetches a webpage (e.g., https://example.com).
b) Run it sequentially and then using ThreadPoolExecutor.
c) Measure the execution time for both approaches and compare performance.
Hints:
• Use requests.get(url) for fetching web pages.
• Use concurrent.futures.ThreadPoolExecutor for threading.
• Fetch 10 different URLs and compare execution times.

Answer:
Link: Assignment_1-2220281-Mst. Aysa Siddika Meem.ipynb

(a) Webpage Fetching Function; I created a simple function using `requests.get(url)` that sends an HTTP request to a given webpage and returns the URL along with its status code. This function is used as the core operation for both the sequential and multithreaded versions.

(b)Sequential vs. Threaded Execution: I then executed the same set of 10 URLs in two ways:

- Sequentially, where each webpage is fetched one after another.
- Using ThreadPoolExecutor, where multiple threads fetch different URLs in parallel. This allowed me to compare the performance difference between normal execution and multithreading

```
Fetching 10 web pages...

Running Sequential Version...
https://example.com -> 200
https://httpbin.org/get -> 200
https://www.python.org -> 200
https://www.github.com -> 200
https://www.wikipedia.org -> 403
https://www.stackoverflow.com -> 403
https://www.openai.com -> 403
https://www.djangoproject.com -> 200
https://www.apple.com -> 200
https://www.microsoft.com -> 200

Sequential Time: 3.9832 seconds

Running Multithreaded Version...
https://example.com -> 200
https://httpbin.org/get -> 200
https://www.python.org -> 200
https://www.github.com -> 200
https://www.wikipedia.org -> 403
https://www.stackoverflow.com -> 403
https://www.openai.com -> 403
https://www.djangoproject.com -> 200
https://www.apple.com -> 200
https://www.microsoft.com -> 200

Threaded Time: 0.9315 seconds
```

(c) Performance comparison and conclusion

For this experiment, I compared the execution time of the sequential and multithreaded versions when fetching 10 different web pages. The sequential version required approximately 3.9832 seconds to complete, while the multithreaded version using `ThreadPoolExecutor` finished in about 0.9315 seconds. Both approaches returned the same HTTP status codes for all URLs, so the correctness of the results is preserved.

These measurements show that the multithreaded implementation is around 4 times faster than the sequential one. The main reason is that this task is I/O-bound: most of the time is spent waiting for network responses rather than using the CPU. With multithreading, several requests can wait for data at the same time, allowing other threads to run instead of the program being idle. Therefore, the results clearly demonstrate that multithreading provides a significant performance benefit for I/O-bound operations such as downloading web pages.

Task 4: Combining Multiprocessing and Multithreading (30 Marks) Extend Task 3 by combining multiprocessing and multithreading:
a) Use multiprocessing to split a list of 100 URLs across 4 processes.
b) Each process will use threading to fetch 10 URLs in parallel.
c) Measure the total execution time and compare it to Task 3.
Hints:
• Use multiprocessing.Pool to distribute the workload.
• Use ThreadPoolExecutor inside each process.

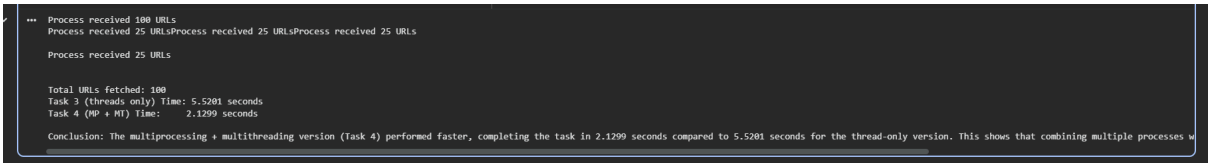
Link: Assignment_1-2220281-Mst. Aysa Siddika Meem.ipynb

(a) Splitting 100 URLs across 4 processes

For this task, I first generated a list of 100 URLs (`https://example.com/?id=0` … `id=99`).Using `multiprocessing.Pool` with 4 processes, I divided this list into 4 chunks of 25 URLs and passed each chunk to a separate worker process. The console output (`Process received 25 URLs`) confirms that each process received exactly one quarter of the total URLs.

(b) Using threading inside each process

Inside each worker process, I used `concurrent.futures.ThreadPoolExecutor` with `max_workers=10` to fetch the URLs assigned to that process. Thus, each process could download up to 10 URLs in parallel, and all four processes were running at the same time. This creates a hybrid model that combines process-level and thread-level parallelism.



(c) Performance measurement and comparison

I measured the total execution time with `time.time()` and compared it to the pure-threading approach from Task 3.

- Task 3 (threads only): 5.5201 seconds
- Task 4 (multiprocessing + multithreading): 2.1299 seconds

Both versions successfully fetched all 100 URLs, but the hybrid version was clearly faster, giving roughly a 2.6× speed-up. This result shows that, for a larger workload, combining multiprocessing with multithreading can further reduce execution time by exploiting multiple CPU processes as well as parallel I/O within each process.

Bonus: Describe when to use Multiprocessing and when to use Multithreading based on your understanding in light of Task3-4. (10 Marks)

Answer:

From the results of Task 3 and Task 4, it becomes clear that multiprocessing and multithreading are suitable for different types of problems. In Task 3, where the program fetched multiple web pages, the work was entirely I/O-bound. The CPU was mostly waiting for network responses, and therefore multithreading performed much faster than the sequential version. Threads allowed many web requests to run in parallel, and the total execution time decreased significantly because threads do not block the CPU during I/O waits.

In contrast, Task 4 involved a larger workload (100 URLs) and combined multiprocessing with multithreading. Multiprocessing created multiple independent processes that shared the workload across CPU cores, while threading inside each process handled the I/O waits. The

hybrid approach performed even better than threads alone because the work was divided across multiple processes, reducing the waiting time further and improving overall throughput. This shows that multiprocessing can be beneficial when handling a large volume of tasks, or when preparing many threads requires CPU coordination.

Based on these observations, multithreading is best used when the task is I/O-bound and involves waiting (e.g., downloading web pages, reading files, API calls). Multiprocessing, on the other hand, is more suitable for CPU-bound or large-scale parallel workloads that must be distributed across CPU cores. A combination of both, as shown in Task 4, is most effective when the system needs to handle large numbers of I/O tasks while also benefiting from process-level parallelism.