Q1. What is multiprocessing in Python? Why is it useful?

Multiprocessing in Python is a technique that allows you to create multiple independent processes, each with its own memory space and resources, to execute code concurrently. It is a way to achieve parallelism in Python, which can utilize multiple CPU cores, leading to improved performance and faster execution of tasks. Multiprocessing is particularly useful for CPU-bound tasks, such as data processing, where you want to take full advantage of the available hardware.

Q2. What are the differences between multiprocessing and multithreading?

Multiprocessing and multithreading are both techniques for achieving concurrency, but they have some key differences:

Processes vs. Threads:

Multiprocessing uses multiple processes, each with its own memory space and resources. Multithreading uses multiple threads within a single process, sharing the same memory space. Isolation:

Multiprocessing provides better isolation between processes, making it more suitable for CPU-bound tasks that might need to run in parallel without interfering with each other. Multithreading is suitable for I/O-bound tasks where threads can block while waiting for I/O operations. However, threads in the same process can interfere with each other's memory. GIL (Global Interpreter Lock):

In CPython (the standard Python interpreter), the Global Interpreter Lock (GIL) allows only one thread to execute Python code at a time. This limits the effectiveness of multithreading for CPU-bound tasks. Multiprocessing does not have the GIL limitation and can fully utilize multiple CPU cores. Complexity:

Multiprocessing can be more complex to work with due to the need to manage inter-process communication and data sharing. Multithreading is generally simpler to work with but can lead to complex synchronization issues.

Q3. Write a Python code to create a process using the multiprocessing module.

In [1]:
```python
import multiprocessing

def worker_function():
    print("Worker process is running.")

if __name__ == "__main__":
    # Create a multiprocessing process
    process = multiprocessing.Process(target=worker_function)

    # Start the process
    process.start()

    # Wait for the process to complete
    process.join()

    print("Main process has finished.")
```

```
Worker process is running.
Main process has finished.
```

Q4. What is a multiprocessing pool in Python? Why is it used?

A multiprocessing pool in Python, often represented by multiprocessing.Pool, is a convenient way to manage and distribute multiple tasks to a pool of worker processes. It simplifies the process of parallelizing tasks by allowing you to submit functions or methods for execution, and the pool takes care of distributing the tasks among the worker processes.

Why it's used:

Pools provide a high-level interface for parallelizing tasks, making it easier to work with multiple processes. They automatically manage the worker processes, distributing tasks, and collecting results. Pools are useful when you have a large number of similar tasks to perform in parallel, such as data processing, web scraping, or simulations.

Q5. How can we create a pool of worker processes in Python using the multiprocessing module?

In [2]:
```
import multiprocessing

def worker_function(x):
    return x * x

if __name__ == "__main__":
    # Create a multiprocessing pool with 4 worker processes
    with multiprocessing.Pool(processes=4) as pool:
        # Distribute tasks to the pool
        result = pool.map(worker_function, [1, 2, 3, 4, 5])

    print(result)
```

```
[1, 4, 9, 16, 25]
```

Q6. Write a Python program to create 4 processes, each process should print a different number using the multiprocessing module in Python.

In [3]:
```
import multiprocessing

def print_number(number):
    print(f"Process {number}: {number}")

if __name__ == "__main__":
    processes = []

    for i in range(1, 5):
        process = multiprocessing.Process(target=print_number, args=(i,))
        processes.append(process)
        process.start()
    for process in processes:
        process.join()

    print("All processes have finished.")
```

```
Process 1: 1
Process 2: 2
Process 3: 3
Process 4: 4
All processes have finished.
```