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

ANS: **Multiprocessing in Python** refers to the capability of the Python interpreter to run multiple processes concurrently. It is a form of parallelism and allows programs to utilize multiple CPU cores effectively.

Why is multiprocessing useful?

1. **Improved Performance**: By utilizing multiple CPU cores, multiprocessing allows programs to execute tasks concurrently, thereby reducing overall execution time for CPU-bound tasks.
2. **Parallelism**: It enables true parallel execution of tasks, which is beneficial for applications that can be split into independent parts that can run simultaneously.
3. **Enhanced CPU Utilization**: Modern computers often have multiple CPU cores. Multiprocessing allows programs to fully utilize these cores, thus maximizing the computing power available.
4. **Scalability**: Multiprocessing facilitates scaling up computational tasks by distributing workload across multiple processes, making it easier to handle larger datasets or more intensive computations.

Q2. What are the differences between multiprocessing and multithreading?

ANS: Execution Model:

- **Multiprocessing**:
  - **Processes**: In multiprocessing, separate processes are created. Each process has its own memory space and runs independently of other processes. Processes do not share memory by default, which can make communication between processes more complex but also avoids issues like shared memory conflicts.
  - **Concurrency**: Processes can execute simultaneously on different CPU cores, achieving true parallelism.
- **Multithreading**:
  - **Threads**: Multithreading involves multiple threads within the same process. Threads share the same memory space (global variables, heap memory, etc.) of the process that created them.
  - **Concurrency**: Threads share CPU resources within a process. They can run concurrently and appear to execute simultaneously, but the actual parallelism depends on the operating system's ability to schedule threads across CPU cores. Python's Global Interpreter Lock (GIL) limits true parallel execution of Python bytecode within a single process, often making multithreading more suitable for I/O-bound tasks rather than CPU-bound tasks.

2. Memory Overhead:

- **Multiprocessing**:

    - Each process has its own memory space, which includes a separate copy of the program's memory. This separation requires more memory compared to multithreading.
    - Inter-process communication (IPC) mechanisms (like queues, pipes, shared memory) are used to exchange data between processes.
- **Multithreading**:
    - Threads share the same memory space, which means they have access to the same variables and data structures. This sharing can simplify communication between threads but requires careful synchronization to avoid race conditions and ensure data integrity.

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

ANS: Creating a process using the multiprocessing module in Python involves defining a target function that the process will execute. Here's a simple example of how to create a process using multiprocessing:

CODE:

```
import multiprocessing

import os

def worker(num):

    """Function to print process ID and a number."""

    print(f"Worker process ID: {os.getpid()}, Number: {num}")

if __name__ == "__main__":

    process = multiprocessing.Process(target=worker, args=(5,))

    process.start()

    process.join()


    print("Main process ID:", os.getpid())

    print("Process finished.")
```

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

ANS: A multiprocessing pool in Python refers to a group of worker processes that are managed together to perform tasks concurrently. It is provided by the multiprocessing.Pool class in Python's multiprocessing module. The primary purpose of using a multiprocessing pool is to efficiently distribute tasks across multiple processes, leveraging the capabilities of multi-core CPUs for parallel execution.

Why is a multiprocessing pool used?

1. **Concurrency and Parallelism**: A multiprocessing pool allows tasks to be executed concurrently across multiple processes, taking advantage of multiple CPU cores. This can significantly speed up the execution of CPU-bound tasks that can be split into independent parts.
2. **Ease of Use**: The multiprocessing pool abstracts away the complexities of managing individual processes. It provides a high-level interface for distributing work and collecting results from multiple processes.
3. **Task Distribution**: Tasks are typically distributed among the processes in the pool using methods like apply(), apply_async(), map(), and map_async(). These methods automatically partition the workload and manage communication between the main program and the worker processes.
4. **Resource Management**: The pool manages the creation and termination of worker processes, optimizing resource usage and ensuring that processes are reused for multiple tasks (if using a persistent pool).

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

ANS: To create a pool of worker processes in Python using the multiprocessing module, you can use the multiprocessing.Pool class. This allows you to manage a group of worker processes that can execute tasks concurrently. Here's how you can create and use a multiprocessing pool:

CODE:

```
import multiprocessing

def calculate_square(number):
    return number * number

if __name__ == "__main__":
    pool = multiprocessing.Pool(processes=3)
    numbers = [1, 2, 3, 4, 5]
    results = pool.map(calculate_square, numbers)
    pool.close()
    pool.join()
    print("Original numbers:", numbers)
    print("Squared numbers:", results)
```

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

ANS: CODE:

```
import multiprocessing

import os

def print_number(number):

    print(f"Process ID: {os.getpid()}, Number: {number}")

if __name__ == "__main__":

    numbers = [1, 2, 3, 4]

    processes = []

    for number in numbers:

        process = multiprocessing.Process(target=print_number, args=(number,))

        processes.append(process)

        process.start()

    for process in processes:

        process.join()

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

Explanation:

1. **Importing Modules**: Importing multiprocessing for process management and os to get process IDs.
2. **Define the Worker Function**: print_number(number) is a function that prints the process ID (os.getpid()) and a given number.
3. **Main Execution Block (if __name__ == "__main__":)**: Ensures that the code inside it runs only when the script is executed directly (not when imported as a module).
4. **Define Tasks**:
    - numbers = [1, 2, 3, 4]: List of numbers for each process to print.

 Create Processes:

- Iterate through the numbers list and create a multiprocessing.Process object for each number. Each process is targeted to execute the print_number function with the corresponding number as an argument.

- **Start Processes**:
  - Call process.start() to start each process concurrently
- **Wait for Processes to Complete**:
  - Use process.join() to wait for each process to finish executing before proceeding.
- **Print Completion Message**:
  - Finally, print a message indicating that all processes have finished.