Q1.
Multiprocessing in Python refers to the capability of executing multiple processes concurrently, each with its own memory space, and typically utilizing multiple CPU cores. It's useful for tasks that can be parallelized, such as CPU-bound operations, because it allows for better utilization of available resources and can significantly improve performance.

Q2.
The main differences between multiprocessing and multithreading are:

Process vs. Thread: In multiprocessing, multiple processes are created, each with its own memory space, while in multithreading, multiple threads exist within the same process and share the same memory space.

Resource Overhead: Multiprocessing generally incurs more overhead in terms of memory and system resources due to the separate memory spaces for each process. Multithreading typically has less overhead since threads share memory.

Concurrency: Multiprocessing is suitable for tasks that are CPU-bound or I/O-bound, whereas multithreading is more suitable for I/O-bound tasks because of the Global Interpreter Lock (GIL) in Python, which prevents multiple threads from executing Python bytecode simultaneously.

Parallelism: Multiprocessing achieves true parallelism by utilizing multiple CPU cores, while multithreading may not fully utilize multiple CPU cores due to the GIL.

Q3. Here's a simple Python code to create a process using the multiprocessing module:

```
import multiprocessing

def worker():
    print("Worker process")

if __name__ == "__main__":
    process = multiprocessing.Process(target=worker)
    process.start()
    process.join()
```

Q4.
A multiprocessing pool in Python, provided by the multiprocessing.Pool class, is a convenient way to distribute tasks across multiple worker processes. It's used to parallelize the execution of a function across multiple input values.

Q5. To create a pool of worker processes in Python using the multiprocessing module, you can use the Pool class. Here's a basic example:

```
\import multiprocessing

def worker(num):
    return num * num

if __name__ == "__main__":
    with multiprocessing.Pool(processes=4) as pool:
        results = pool.map(worker, range(10))
    print(results)
```

Q6. Here's a Python program to create 4 processes, each printing a different number using the multiprocessing module:

```
import multiprocessing

def print_number(num):
    print("Process ID:", multiprocessing.current_process().pid, "-
Number:", num)

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()
```

This code creates 4 processes, each printing a different number. Each process has its own process ID obtained using multiprocessing.current_process().pid.