# Multiprocessing In Python

Bhimashankar Takalki

Multiprocessing in Python refers to the ability to run multiple processes concurrently, utilizing multiple CPU cores to perform tasks in parallel. This is particularly useful for CPU-bound tasks that can benefit from parallel execution. Python provides the multiprocessing module for working with multiple processes.

```
import multiprocessing

def worker():
    """Function to be executed by each process"""
    print("Worker function executing")

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

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

    print("Main process exiting")
```

In this example:

- We define a worker function that represents the task to be executed by each process.
- We create a multiprocessing.Process object and specify the target argument as the worker function.
- We start the process using the start() method.
- We wait for the process to finish using the join() method.
- Finally, we print a message indicating that the main process is exiting.

**Passing Arguments to Processes:**

```
import multiprocessing

def worker(num):
    """Function to be executed by each process"""
    print(f"Worker function executing with argument: {num}")

if __name__ == "__main__":
    # Create and start a process with an argument
    process = multiprocessing.Process(target=worker, args=(10,))
    process.start()
    process.join()

    print("Main process exiting")
```

A Pool of Processes in Python, specifically in the context of the `multiprocessing` module, refers to a group of worker processes that are created and managed together to perform concurrent tasks. The `multiprocessing.Pool` class provides a convenient way to manage a pool of worker processes and distribute tasks among them.

Here's an overview of how a pool of processes works:

1. **Creation of Worker Processes**: When you create a `multiprocessing.Pool` object, it automatically creates a specified number of worker processes. These worker processes are separate from the main process and run concurrently.
2. **Distribution of Tasks**: You can use various methods of the `multiprocessing.Pool` class, such as `map()`, `apply()`, `apply_async()`, etc., to distribute tasks among the worker processes. These methods allow you to execute functions or methods with arguments across multiple processes concurrently.
3. **Execution of Tasks**: Once tasks are distributed to the worker processes, they execute the assigned tasks concurrently. Each worker process runs independently and performs its assigned task.

4. **Result Gathering**: After the worker processes finish executing their tasks, the results (if any) are collected and returned to the main process. This allows you to retrieve the results of the concurrent tasks and continue with further processing as needed.
5. **Cleanup**: Once all tasks are completed and results are collected, the worker processes are terminated, and any resources associated with the pool of processes are released.

Using a pool of processes can significantly improve the performance of CPU-bound tasks by leveraging multiple CPU cores and executing tasks concurrently.

## Pool of Processes:

```
import multiprocessing

def worker(num):
    """Function to be executed by each process"""
    return num * 2

if __name__ == "__main__":
    # Create a pool of processes
    pool = multiprocessing.Pool(processes=4)

    # Map the worker function to multiple arguments
    results = pool.map(worker, [1, 2, 3, 4, 5])

    # Close the pool and wait for all processes to finish
    pool.close()
    pool.join()

    print("Results:", results)
```