استفاده از Multiprocessing برای انجام پردازش‌های موازی

در پایتون، برای انجام پردازش‌های موازی (parallel processing)، دو روش اصلی وجود دارد: **Threading** و **Multiprocessing**. این دو تکنیک متفاوت هستند و به روش‌های مختلفی برای استفاده از منابع سیستم بهره می‌برند.

1. تفاوت بین Multiprocessing و Threading در پایتون

- Threading
  :
  - در **Threading** چندین **Thread** در همان فرآیند (Process) اجرا می‌شوند و از یک فضای حافظه مشترک استفاده می‌کنند.
  - این روش برای کارهای I/O-bound (که بیشتر وابسته به زمان انتظار هستند، مانند خواندن/نوشتن فایل‌ها، اتصال به شبکه و ...) مناسب است.
  - در پایتون، به دلیل وجود **GIL (Global Interpreter Lock)**، عملکرد **Threading** در پردازش‌های **CPU-bound** (که نیاز به محاسبات سنگین دارند) محدود است.
- Multiprocessing
  :
  - در **Multiprocessing** هر **Process** فضای حافظه مجزا و اختصاصی خود را دارد.
  - این روش برای پردازش‌های **CPU-bound** مناسب است زیرا هر فرآیند به طور مستقل از دیگر فرآیندها اجرا می‌شود و می‌تواند از چندین هسته پردازشی (core) استفاده کند.
  - با استفاده از **multiprocessing**، مشکل **GIL** در پایتون برطرف می‌شود.

2. ایجاد و مدیریت `Process`ها با استفاده از کلاس `Process` در کتابخانه `multiprocessing`

در پایتون می‌توان با استفاده از کلاس `Process` از کتابخانه `multiprocessing` برای ایجاد پردازش‌ها استفاده کرد.

مثال:

```
import multiprocessing
import time

# تابعی که قرار است در هر پردازش اجرا شود
def worker(num):
    print(f"Process {num} is starting")
    time.sleep(2)
    print(f"Process {num} has finished")

# ایجاد و راه‌اندازی پردازش‌ها
if __name__ == "__main__":
    processes = []
    for i in range(3):
        p = multiprocessing.Process(target=worker, args=(i,))
        processes.append(p)
        p.start()

    # منتظر ماندن تا پایان تمام پردازش‌ها
```

```
    for p in processes:
        p.join()

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

در اینجا:

- تابع `worker` در هر پردازش اجرا می‌شود.
- با استفاده از `()start` پردازش‌ها شروع می‌شوند و با `()join` منتظر می‌مانیم تا تمام پردازش‌ها به اتمام برسند.

3. استفاده از `Queue` و `Pipe` برای ارسال داده بین Processها

برای ارتباط بین پردازش‌ها، می‌توان از **Queue** و **Pipe** استفاده کرد.

- **Queue**: یک صف (queue) است که می‌توان داده‌ها را از یک پردازش به پردازش دیگر ارسال کرد.
- **Pipe**: یک کانال ارتباطی دوطرفه است که بین دو پردازش استفاده می‌شود.

مثال با استفاده از `Queue`:

```
import multiprocessing

def worker(q):
    q.put('Hello from process')

if __name__ == "__main__":
    q = multiprocessing.Queue()

    # ایجاد پردازش و شروع آن
    p = multiprocessing.Process(target=worker, args=(q,))
    p.start()

    # دریافت داده از Queue
    print(q.get())  # خروجی: Hello from process

    p.join()
```

در اینجا:

- پردازش `worker` داده‌ای را در `Queue` قرار می‌دهد و پردازش اصلی داده‌ها را دریافت می‌کند.

مثال با استفاده از `Pipe`:

```
import multiprocessing

def worker(conn):
    conn.send('Hello from process')
    conn.close()

if __name__ == "__main__":
    parent_conn, child_conn = multiprocessing.Pipe()

    # ایجاد پردازش و شروع آن
    p = multiprocessing.Process(target=worker, args=(child_conn,))
```

```
    p.start()

    # دریافت داده از Pipe
    print(parent_conn.recv())  # خروجی: Hello from process

    p.join()
```

در اینجا:

- `Pipe` دو کانال ارتباطی (یک برای ارسال و یکی برای دریافت داده) ایجاد می‌کند.

4. استفاده از `Pool` برای اجرای پردازش‌ها به‌صورت موازی در مجموعه‌ای از پردازش‌ها

برای پردازش‌های موازی در مجموعه‌ای از داده‌ها، می‌توان از **Pool** استفاده کرد. این ابزار به شما این امکان را می‌دهد که پردازش‌ها را به صورت گروهی مدیریت کنید و برای هر داده یک پردازش جداگانه ایجاد کنید.

مثال:

```
import multiprocessing

def square(x):
    return x * x

if __name__ == "__main__":
    # ایجاد Pool با تعداد پردازش‌ها
    with multiprocessing.Pool(processes=4) as pool:
        result = pool.map(square, [1, 2, 3, 4, 5])

    print(result)  # خروجی: [1, 4, 9, 16, 25]
```

در اینجا:

- `pool.map()` مشابه `map()` در پایتون است، با این تفاوت که هر عنصر به پردازشی جداگانه اختصاص داده می‌شود.
- `Pool` به صورت خودکار پردازش‌ها را ایجاد و مدیریت می‌کند.

5. مقایسه عملکرد و بهینه‌سازی استفاده از `Threading` و `Multiprocessing` در پایتون

زمان اجرا:

- در پردازش‌های **I/O-bound** مانند خواندن فایل‌ها یا اتصال به سرور، استفاده از **Threading** می‌تواند مفیدتر باشد، زیرا تعداد Threadها می‌توانند منتظر I/O شوند بدون اینکه پردازش‌های دیگر متوقف شوند.
- در پردازش‌های **CPU-bound** که نیاز به محاسبات سنگین دارند، **Multiprocessing** گزینه بهتری است زیرا می‌تواند از چندین هسته پردازشی به‌طور هم‌زمان استفاده کند و مشکل **GIL** (Global Interpreter Lock) را دور بزند.

کاهش Overhead:

- ایجاد **Thread**ها در مقایسه با **Process**ها هزینه کمتری دارد زیرا همه Threadها در فضای حافظه یکسانی اجرا می‌شوند، در حالی که پردازش‌ها فضای حافظه اختصاصی دارند.
- برای پردازش‌های با نیاز بالا به منابع سیستم، **Multiprocessing** به مراتب مناسب‌تر است.

محدودیت‌های GIL:

- پایتون **GIL** را برای هماهنگی Threadها استفاده می‌کند، که به پردازش‌های **CPU-bound** آسیب می‌زند.
- **Multiprocessing** به دلیل اینکه هر پردازش فضای حافظه مجزای خود را دارد، می‌تواند پردازش‌ها را به طور واقعی موازی اجرا کند.

6. نتیجه‌گیری

- **Threading** بیشتر برای **I/O-bound** مناسب است، در حالی که **Multiprocessing** برای **CPU-bound** مناسب‌تر است.
- **Multiprocessing** به شما این امکان را می‌دهد که از پردازش‌های واقعی موازی استفاده کنید و می‌تواند از چندین هسته پردازشی بهره‌برداری کند.
- برای بهینه‌سازی پردازش‌های موازی، می‌توان از **Queue** و **Pipe** برای ارتباط بین پردازش‌ها و از **Pool** برای مدیریت پردازش‌ها به‌صورت گروهی استفاده کرد.