استفاده از `multiprocessing` و `threading` در پایتون

در پایتون، برای انجام پردازش‌های موازی و همزمان، دو ابزار اصلی وجود دارند: `threading` و `multiprocessing`. هرکدام از این ابزارها مزایا و محدودیت‌های خود را دارند، و انتخاب مناسب‌ترین روش بستگی به نوع کار مورد نظر دارد. در اینجا به تفاوت‌ها، مزایا و محدودیت‌های این دو ابزار پرداخته و روش‌های بهینه‌سازی برنامه‌ها برای پردازش‌های موازی را بررسی می‌کنیم.

تفاوت‌های بین `threading` و `multiprocessing`

1. Threading

- **تعریف**: `threading` به شما این امکان را می‌دهد که چندین نخ (thread) را در یک پردازش (process) اجرا کنید. هر نخ می‌تواند یک وظیفه خاص را انجام دهد.
- **محدودیت GIL**: در پایتون، گلوبال اینترپرتور لاک (GIL) وجود دارد که تنها به یک نخ در هر زمان اجازه می‌دهد که کد را اجرا کند. این به این معناست که `threading` برای پردازش‌های **CPU-bound** (که نیاز به پردازش سنگین دارند) کارایی کمی خواهد داشت.
- مزایا
  :
  - برای برنامه‌هایی که نیاز به انجام کارهای ورودی/خروجی (I/O-bound) دارند، مانند درخواست‌های HTTP یا خواندن/نوشتن فایل‌ها، مناسب است.
  - مصرف حافظه کمتری نسبت به `multiprocessing` دارد، چون همه نخ‌ها در یک فضای حافظه مشترک اجرا می‌شوند.
- معایب
  :
  - به دلیل GIL، در پردازش‌های **CPU-bound** عملکرد خوبی ندارد.

2. Multiprocessing

- **تعریف**: `multiprocessing` به شما این امکان را می‌دهد که چندین پردازش (process) را همزمان اجرا کنید. هر پردازش فضای حافظه و منابع جداگانه‌ای دارد.
- **عدم وجود GIL**: به دلیل اینکه هر پردازش مستقل است، هیچ مشکلی از نظر GIL وجود ندارد و پردازش‌های **CPU-bound** می‌توانند به‌طور واقعی موازی اجرا شوند.
- مزایا
  :
  - برای پردازش‌های **CPU-bound** (مانند الگوریتم‌های پیچیده ریاضی یا پردازش‌های سنگین) مناسب است، چون هر پردازش می‌تواند از یک هسته جداگانه CPU استفاده کند.
  - به دلیل جدا بودن پردازش‌ها، هر پردازش می‌تواند به‌طور مستقل از هم دیگر اجرا شود.
- معایب
  :
  - مصرف حافظه بیشتری دارد، چون هر پردازش فضای حافظه جداگانه‌ای دارد.
  - ایجاد و مدیریت پردازش‌ها نسبت به نخ‌ها پیچیده‌تر است.

انتخاب مناسب‌ترین روش برای پردازش‌های موازی

- اگر کار شما بیشتر مربوط به ورودی/خروجی (I/O-bound) است

  (مانند خواندن فایل‌ها یا ارسال درخواست‌های HTTP):

  - `threading` می‌تواند انتخاب بهتری باشد. نخ‌ها می‌توانند منتظر تکمیل عملیات I/O بمانند و در این حین از پردازنده استفاده بهینه کنند.
- اگر کار شما بیشتر مربوط به پردازش‌های سنگین و پیچیده (CPU-bound) است

  (مانند انجام محاسبات ریاضی پیچیده یا پردازش داده‌های بزرگ):

  - `multiprocessing` مناسب‌تر است، زیرا این روش می‌تواند از چندین هسته CPU به‌طور همزمان استفاده کند و عملکرد بهتری در پردازش‌های سنگین خواهد داشت.

بهینه‌سازی برنامه‌ها برای انجام چندین پردازش به طور همزمان

1. استفاده از `Queue` برای تبادل داده‌ها بین پردازش‌ها (Multiprocessing)

زمانی که از `multiprocessing` استفاده می‌کنید، ممکن است نیاز داشته باشید که داده‌ها را بین پردازش‌ها منتقل کنید. یکی از راه‌های انجام این کار استفاده از `Queue` است.

مثال:

```
import multiprocessing

def worker(queue):
    result = "نتیجه پردازش"
    queue.put(result)

if __name__ == '__main__':
    queue = multiprocessing.Queue()
    process = multiprocessing.Process(target=worker, args=(queue,))
    process.start()
    process.join()
    print(queue.get())  # دریافت نتیجه از Queue
```

2. استفاده از `ThreadPoolExecutor` و `ProcessPoolExecutor` برای مدیریت نخ‌ها و پردازش‌ها

در صورتی که نیاز به مدیریت تعدادی نخ یا پردازش دارید، استفاده از `ThreadPoolExecutor` (برای نخ‌ها) و `ProcessPoolExecutor` (برای پردازش‌ها) می‌تواند به شما کمک کند تا به‌طور خودکار نخ‌ها یا پردازش‌ها را مدیریت کنید.

مثال:

```python
from concurrent.futures import ThreadPoolExecutor

def task(x):
    return x * 2

with ThreadPoolExecutor(max_workers=5) as executor:
    results = list(executor.map(task, range(10)))

print(results)
```

3. استفاده از `asyncio` در ترکیب با `multiprocessing`

در برخی سناریوها می‌توان از ترکیب `asyncio` و `multiprocessing` استفاده کرد. برای مثال، در صورتی که نیاز به انجام عملیات I/O همزمان با پردازش‌های سنگین دارید، می‌توانید از `asyncio` برای مدیریت عملیات I/O و از `multiprocessing` برای پردازش‌های سنگین استفاده کنید.

مثال ترکیبی:

```python
import asyncio
import multiprocessing

def cpu_bound_task(x):
    return x * 2

async def io_bound_task(x):
    await asyncio.sleep(1)
    return x * 3

async def main():
    loop = asyncio.get_event_loop()
    with multiprocessing.Pool(processes=2) as pool:
        cpu_results = await loop.run_in_executor(None, pool.map, cpu_bound_task, range(5))
        io_results = await asyncio.gather(*[io_bound_task(x) for x in range(5)])
    print(cpu_results)
    print(io_results)

asyncio.run(main())
```

نتیجه‌گیری

- `threading` برای برنامه‌های I/O-bound (مانند درخواست‌های HTTP، خواندن فایل‌ها و ...) مفید است.
- `multiprocessing` برای پردازش‌های سنگین CPU-bound که به پردازش موازی واقعی نیاز دارند مناسب‌تر است.
- ترکیب این دو با `asyncio` می‌تواند برای سناریوهایی که نیاز به هر دو نوع عملیات دارند، بهترین گزینه باشد.

بهینه‌سازی استفاده از این ابزارها بستگی به نوع برنامه و نیازهای خاص شما دارد.