برنامه‌نویسی غیرهمزمان با `asyncio`

در پایتون، **برنامه‌نویسی غیرهمزمان** به معنای اجرای عملیات به طور همزمان و بدون انتظار برای اتمام هر عملیات است. این تکنیک به‌ویژه برای کارهایی که بیشتر به **I/O (ورودی/خروجی)** وابسته‌اند، مانند خواندن یا نوشتن داده‌ها به فایل، شبکه و یا دیتابیس، مناسب است. استفاده از برنامه‌نویسی غیرهمزمان به بهینه‌سازی استفاده از منابع سیستم و بهبود کارایی کمک می‌کند.

1. مفهوم برنامه‌نویسی غیرهمزمان و تفاوت آن با برنامه‌نویسی همزمان

- **برنامه‌نویسی همزمان (Synchronous Programming)**:
  - در برنامه‌نویسی همزمان، دستورات به ترتیب اجرا می‌شوند و منتظر می‌مانند تا هر عملیات تکمیل شود.
  - برای مثال، اگر در یک برنامه همزمان بخواهید داده‌هایی را از یک فایل بخوانید، برنامه تا زمانی که داده‌ها خوانده نشوند، متوقف می‌ماند و دیگر کدها اجرا نمی‌شوند.
- **برنامه‌نویسی غیرهمزمان (Asynchronous Programming)**:
  - در برنامه‌نویسی غیرهمزمان، عملیات‌های مختلف به طور همزمان (به صورت موازی) اجرا می‌شوند، بدون اینکه یکی از آن‌ها منتظر دیگری بماند.
  - برای مثال، در برنامه غیرهمزمان می‌توان در هنگام خواندن داده‌ها از یک فایل، سایر بخش‌های برنامه را اجرا کرد.

2. معرفی کتابخانه `asyncio` برای مدیریت عملیات غیرهمزمان

کتابخانه `asyncio` در پایتون ابزاری برای برنامه‌نویسی غیرهمزمان است که برای مدیریت عملیات I/O در پایتون طراحی شده است. این کتابخانه به شما امکان می‌دهد که عملیات‌هایی مانند اتصال به سرورها، خواندن از فایل‌ها و انجام محاسبات را به صورت غیرهمزمان و بدون مسدود کردن برنامه اجرا کنید.

ویژگی‌ها:

- **Event Loop**: مدیریت اجرا و زمان‌بندی وظایف.
- **Tasks**: نمایندگان غیرهمزمان که نمایانگر اجرای یک وظیفه هستند.
- **Coroutines**: توابعی که می‌توانند به صورت غیرهمزمان اجرا شوند.

3. تعریف `Event Loop` و نحوه اجرای آن

`Event Loop` هسته اصلی برنامه‌های غیرهمزمان است. این حلقه وظیفه مدیریت و ترتیب‌دهی به وظایف غیرهمزمان را بر عهده دارد. وقتی از یک `async function` (یا همان coroutine) استفاده می‌کنید، این توابع به رویدادهای غیرهمزمان تبدیل می‌شوند که توسط `Event Loop` اجرا می‌شوند.

نحوه اجرای `Event Loop`:

- `Event Loop` به طور مداوم منتظر می‌ماند تا یک رویداد اتفاق بیافتد (مثلاً دریافت پاسخ از سرور).
- زمانی که رویدادی اتفاق می‌افتد، این رویداد در `Event Loop` قرار می‌گیرد و سپس اجرا می‌شود.

```
import asyncio

async def main():
    print("Hello")
    await asyncio.sleep(1)  # وظیفه غیرهمزمان، مشابه خوابیدن برای 1 ثانیه
    print("World")

# اجرای Event Loop
asyncio.run(main())
```

در اینجا:

- `main()` یک **coroutine** است که عملیات‌های غیرهمزمان انجام می‌دهد.
- `await asyncio.sleep(1)` از آن برای تعلیق غیرهمزمان استفاده می‌کنیم، تا نشان دهیم که در این زمان، برنامه دیگر متوقف نشده و می‌تواند کارهای دیگر را انجام دهد.
- `asyncio.run(main())` به صورت خودکار یک `Event Loop` را راه‌اندازی و آن را اجرا می‌کند.

4. تفاوت بین `Multiprocessing`, `Threading` و `Asyncio` در مدیریت همزمانی

- **Threading**:
  - در **Threading**، چندین Thread در یک **Process** اجرا می‌شوند. این روش برای پردازش‌های I/O-bound مناسب است، اما به دلیل وجود **GIL** در پایتون، برای پردازش‌های **CPU-bound** بهینه نیست.
  - Threadها به طور همزمان در حافظه مشترک یک فرآیند اجرا می‌شوند.
- **Multiprocessing**:
  - **Multiprocessing** استفاده از چندین **Process** را برای انجام پردازش‌های موازی مدیریت می‌کند. هر پردازش فضای حافظه مجزای خود را دارد.
  - این تکنیک برای پردازش‌های **CPU-bound** بسیار مفید است.
- **Asyncio**:
  - در **Asyncio**، چندین **coroutine** به صورت غیرهمزمان اجرا می‌شوند و از یک **Event Loop** برای مدیریت زمان‌بندی این وظایف استفاده می‌شود.
  - این روش برای کارهای **I/O-bound** مناسب است و به شما این امکان را می‌دهد که بدون استفاده از Thread یا Processهای اضافی، عملیات‌های همزمان را مدیریت کنید.

5. استفاده از `asyncio.run()` برای اجرای برنامه‌های غیرهمزمان

در پایتون، برای اجرای یک برنامه غیرهمزمان، از `asyncio.run()` استفاده می‌کنیم. این تابع به طور خودکار یک **Event Loop** را شروع کرده، برنامه را اجرا می‌کند و سپس **Event Loop** را پس از اتمام اجرا می‌بندد.

مثال:

```
import asyncio

async def fetch_data():
    print("Fetching data...")
    await asyncio.sleep(2)
    return "Data fetched"
```

```
async def process_data():
    print("Processing data...")
    await asyncio.sleep(1)
    return "Data processed"

async def main():
    task1 = asyncio.create_task(fetch_data())
    task2 = asyncio.create_task(process_data())

    result1 = await task1
    result2 = await task2

    print(result1)
    print(result2)

# اجرای برنامه غیرهمزمان
asyncio.run(main())
```

در اینجا:

- `asyncio.create_task()` برای ساخت و اجرای توابع غیرهمزمان استفاده می‌شود.
- `await` برای انتظار و دریافت نتایج از توابع غیرهمزمان است.
- `asyncio.run(main())` برای اجرای `main()` به صورت غیرهمزمان است.

نتیجه‌گیری

- **برنامه‌نویسی غیرهمزمان** به ما این امکان را می‌دهد که عملیات‌های طولانی‌مدت مانند I/O را بدون توقف دیگر بخش‌های برنامه انجام دهیم.
- `asyncio` ابزاری قدرتمند برای مدیریت عملیات غیرهمزمان است و استفاده از آن در پروژه‌های I/O-bound می‌تواند عملکرد بهتری نسبت به `Threading` یا `Multiprocessing` داشته باشد.
- تفاوت‌های اصلی بین **Threading**, **Multiprocessing** و **Asyncio** در نحوه اجرای همزمانی و نوع کاربرد آن‌ها است.