استفاده از منابع همزمان با مدیریت امنیت (با استفاده از `asyncio.Lock` و `asyncio.Semaphore`)

در برنامه‌نویسی غیرهمزمان، یکی از چالش‌های مهم مدیریت همزمانی (concurrency) و دسترسی به منابع مشترک است. این چالش می‌تواند منجر به مشکلاتی مانند **race condition** (شرایط رقابتی) شود که در آن دو یا چند وظیفه به طور همزمان به یک منبع مشترک دسترسی دارند و باعث رفتار غیرمنتظره یا خطا می‌شود.

برای جلوگیری از این مشکلات و هماهنگ‌سازی دسترسی به منابع، می‌توان از ابزارهای همزمانی مختلف مانند `asyncio.Lock` و `asyncio.Semaphore` استفاده کرد.

1. مدیریت Race Condition با `asyncio.Lock`

`asyncio.Lock` به شما اجازه می‌دهد که تنها یک task در هر زمان به یک بخش کد یا منبع مشترک دسترسی داشته باشد. این قفل (lock) برای جلوگیری از دسترسی همزمان به منابع مشترک استفاده می‌شود. وقتی یک task به قفل دسترسی پیدا می‌کند، سایر taskها باید منتظر بمانند تا قفل آزاد شود.

مثال استفاده از `asyncio.Lock`:

```
import asyncio

# تعریف قفل برای مدیریت دسترسی همزمان
lock = asyncio.Lock()

async def access_shared_resource(name):
    async with lock:  # استفاده از قفل برای جلوگیری از دسترسی همزمان
        print(f"{name} is accessing the shared resource.")
        await asyncio.sleep(1)  # شبیه‌سازی عملیات غیرهمزمان
        print(f"{name} has finished using the resource.")

async def main():
    # اجرای چندین task که به منبع مشترک دسترسی دارند
    await asyncio.gather(
        access_shared_resource("Task A"),
        access_shared_resource("Task B"),
        access_shared_resource("Task C")
    )

asyncio.run(main())
```

در این مثال:

- `async with lock` به هر task اجازه می‌دهد که تنها زمانی که قفل آزاد است، به منبع مشترک دسترسی پیدا کند.
- تنها یک task می‌تواند به منبع مشترک دسترسی داشته باشد و بقیه باید منتظر بمانند.

2. مدیریت دسترسی محدود با `asyncio.Semaphore`

`asyncio.Semaphore` شبیه به قفل است، اما این ابزار به شما این امکان را می‌دهد که تعداد معینی از taskها همزمان به منبع مشترک دسترسی پیدا کنند. این ابزار مفید است وقتی که بخواهید تعداد مشخصی از taskها را اجازه دهید که همزمان به منبع دسترسی داشته باشند.

مثال استفاده از `asyncio.Semaphore`:

```python
import asyncio

# تعریف یک Semaphore برای دسترسی همزمان با حداکثر تعداد 2
semaphore = asyncio.Semaphore(2)

async def access_limited_resource(name):
    async with semaphore:  # محدود کردن تعداد دسترسی‌ها با Semaphore
        print(f"{name} is accessing the limited resource.")
        await asyncio.sleep(2)  # شبیه‌سازی عملیات غیرهمزمان
        print(f"{name} has finished using the resource.")

async def main():
    # اجرای چندین task که به منابع محدود دسترسی دارند
    await asyncio.gather(
        access_limited_resource("Task A"),
        access_limited_resource("Task B"),
        access_limited_resource("Task C"),
        access_limited_resource("Task D")
    )

asyncio.run(main())
```

در این مثال:

- `asyncio.Semaphore(2)` به این معنی است که تنها دو task می‌توانند به طور همزمان به منبع مشترک دسترسی پیدا کنند.
- وقتی دو task در حال استفاده از منبع مشترک هستند، taskهای بعدی باید منتظر بمانند تا یک نفر از منبع استفاده نکرده و قفل Semaphore آزاد شود.

3. مقایسه `asyncio.Lock` و `asyncio.Semaphore`

- `asyncio.Lock` برای مدیریت دسترسی انحصاری به یک منبع است. تنها یک task می‌تواند در هر زمان به منبع دسترسی داشته باشد.
- `asyncio.Semaphore` به شما این امکان را می‌دهد که تعداد معینی از taskها همزمان به یک منبع دسترسی پیدا کنند، که برای منابع محدودتر مناسب است.

نتیجه‌گیری

برای جلوگیری از **race condition** و هماهنگ‌سازی دسترسی به منابع در برنامه‌های غیرهمزمان، از `asyncio.Lock` و `asyncio.Semaphore` می‌توان استفاده کرد:

- `asyncio.Lock` برای جلوگیری از دسترسی همزمان به یک منبع استفاده می‌شود.
- `asyncio.Semaphore` برای محدود کردن تعداد taskهایی که می‌توانند به طور همزمان به منبع دسترسی پیدا کنند، مفید است.

استفاده از این ابزارها باعث می‌شود که برنامه‌های غیرهمزمان به طور ایمن و کارآمدتر با منابع مشترک کار کنند.