Pemrograman Multithreading dengan Python

Apa itu *Thread*?

- **Thread** adalah unit eksekusi terkecil dalam sebuah program.
- Satu proses dapat memiliki banyak *thread* yang berjalan secara paralel.
- Thread berbagi **memori** dan **resource** dengan proses induknya.

Kenapa Perlu Multithreading?

- Untuk menyelesaikan tugas yang dapat dilakukan secara **paralel**.
- Meningkatkan efisiensi pada program yang menunggu **I/O** (misalnya membaca file, akses internet, komunikasi jaringan).
- Membuat program lebih **responsif** (misalnya aplikasi GUI atau server).

Proses vs Thread

Aspek	Proses	Thread
Memori	Memiliki ruang memori sendiri	Berbagi memori dengan proses
Pembuatan	Lebih berat	Lebih ringan
Komunikasi	Lebih kompleks (IPC)	Mudah (karena berbagi memori)

Global Interpreter Lock (GIL)

- Python memiliki **GIL (Global Interpreter Lock)**, yang membatasi eksekusi hanya **satu thread Python** pada satu waktu.
- Namun, multithreading tetap bermanfaat untuk **I/O-bound task** (misalnya membaca file, network request, komunikasi socket).
- Untuk **CPU-bound task**, biasanya lebih baik menggunakan **multiprocessing**.

a. Membuat Thread dengan `threading.Thread`

```python
import threading
import time

def tugas(nama):
    for i in range(5):
        print(f"Thread {nama}: langkah {i}")
        time.sleep(1)

# Membuat thread
t1 = threading.Thread(target=tugas, args=("A",))
t2 = threading.Thread(target=tugas, args=("B",))

# Menjalankan thread
t1.start()
t2.start()

# Menunggu thread selesai
t1.join()
t2.join()

print("Semua thread selesai.")
```

b. Thread dengan Subclass

```
class MyThread(threading.Thread):
    def __init__(self, nama):
        super().__init__()
        self.nama = nama

    def run(self):
        for i in range(3):
            print(f"Thread {self.nama} sedang berjalan")
            time.sleep(1)

# Membuat dan menjalankan thread
t1 = MyThread("X")
t2 = MyThread("Y")

t1.start()
t2.start()

t1.join()
t2.join()
```

c. Sinkronisasi Thread (Lock)

```
saldo = 0
lock = threading.Lock()

def tambah_saldo():
    global saldo
    for _ in range(100000):
        with lock:
            saldo += 1

t1 = threading.Thread(target=tambah_saldo)
t2 = threading.Thread(target=tambah_saldo)

t1.start()
t2.start()
t1.join()
t2.join()

print("Saldo akhir:", saldo)
```

Contoh Kasus: Download Data dari Beberapa Website

Tanpa Multithreading

```
import time
import requests

urls = [
    "https://httpbin.org/delay/2",
    "https://httpbin.org/delay/2",
    "https://httpbin.org/delay/2",
]

start = time.time()
for url in urls:
    response = requests.get(url)
    print(f"Selesai download {url}")

print("Waktu total:", time.time() - start)
```

Dengan Multithreading

```python
import threading
import time
import requests

urls = [
    "https://httpbin.org/delay/2",
    "https://httpbin.org/delay/2",
    "https://httpbin.org/delay/2",
]

def download(url):
    response = requests.get(url)
    print(f"Selesai download {url}")

start = time.time()

threads = []
for url in urls:
    t = threading.Thread(target=download, args=(url,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("Waktu total:", time.time() - start)
```

5. Ringkasan Materi

- **Thread** adalah unit eksekusi dalam sebuah proses.
- Multithreading di Python cocok untuk **I/O-bound task**.
- Gunakan `threading.Thread` untuk membuat dan menjalankan thread.
- Gunakan **Lock** untuk menghindari race condition.
- Contoh riil: **parallel web scraping / download data** lebih cepat dibandingkan proses sekuensial.