ایجاد و مدیریت Thread

در برنامه‌نویسی، **Thread** و **Process** هر دو برای انجام وظایف هم‌زمان (concurrent) استفاده می‌شوند، اما تفاوت‌هایی بین آنها وجود دارد که در ادامه توضیح داده می‌شود.

1. تعریف و تفاوت بین Thread و Process

- Process
  :
  - یک **Process** یک برنامه مجزا در حافظه است که مستقل از سایر فرآیندها اجرا می‌شود.
  - هر فرآیند فضای حافظه مستقل خود را دارد و معمولاً زمانی که یک فرآیند نیاز به اشتراک‌گذاری منابع (مثل حافظه) با دیگر فرآیندها داشته باشد، باید از مکانیزم‌های بین‌فرآیندی مانند **IPC (Inter-process communication)** استفاده کند.
- Thread
  :
  - یک **Thread** واحدی از یک فرآیند است که در همان فضای حافظه به‌طور مشترک اجرا می‌شود.
  - Threads یک برنامه می‌توانند با هم به‌طور هم‌زمان اجرا شوند و به‌اشتراک‌گذاری داده‌ها و منابع در همان فضای حافظه بپردازند.
  - هر Thread می‌تواند به‌طور مستقل در فرآیند خود اجرا شود و به منابعی مثل داده‌های مشترک دسترسی پیدا کند.

تفاوت‌های اصلی:

- **Process**: فضای حافظه مجزا، هزینه بیشتر برای ایجاد و مدیریت.
- **Thread**: فضای حافظه مشترک، هزینه کمتر برای ایجاد و مدیریت.

2. ایجاد و راه‌اندازی Threadها با استفاده از کتابخانه `threading`

در پایتون برای کار با Threadها از کتابخانه `threading` استفاده می‌شود. در اینجا چگونگی ایجاد و راه‌اندازی یک Thread را توضیح می‌دهیم.

2.1. ایجاد Thread

برای ایجاد یک Thread، می‌توانید از کلاس `Thread` در کتابخانه `threading` استفاده کنید.

```
import threading

# تعریف تابعی که توسط Thread اجرا خواهد شد
def my_function():
    print("Thread is running!")

# ایجاد یک Thread
thread = threading.Thread(target=my_function)

# شروع Thread
thread.start()

# منتظر ماندن تا پایان اجرای Thread
```

```
thread.join()

print("Thread has finished!")
```

در این مثال:

- تابع `my_function` توسط Thread اجرا می‌شود.
- `()start` برای شروع اجرای Thread و `()join` برای منتظر ماندن تا اتمام اجرای آن استفاده می‌شود.

3. استفاده از متدهای `join`، `()start`، و `()is_alive`

- `()start` : این متد برای شروع اجرای یک Thread استفاده می‌شود.
- `()join` : با این متد می‌توان از برنامه خواست که منتظر بماند تا اجرای Thread خاتمه یابد.
- `()is_alive` : این متد بررسی می‌کند که آیا Thread هنوز در حال اجراست یا خیر.

مثال:

```
import threading
import time

def task():
    print("Task started!")
    time.sleep(2)
    print("Task finished!")

# ایجاد و شروع Thread
t = threading.Thread(target=task)
t.start()

# بررسی وضعیت Thread
if t.is_alive():
    print("Thread is still running...")

# منتظر ماندن تا پایان اجرای Thread
t.join()
print("Thread execution completed.")
```

4. استفاده از `ThreadPoolExecutor` برای مدیریت مجموعه‌ای از Threadها

برای مدیریت مجموعه‌ای از Threadها به‌طور هم‌زمان، می‌توان از `ThreadPoolExecutor` که در کتابخانه `concurrent.futures` قرار دارد استفاده کرد. این ابزار مدیریت و اجرا کردن چندین Thread را ساده‌تر می‌کند.

مثال:

```
from concurrent.futures import ThreadPoolExecutor

# تعریف تابعی که توسط Threadها اجرا می‌شود
def my_task(n):
    print(f"Task {n} started!")
    time.sleep(2)
    print(f"Task {n} finished!")

# ایجاد ThreadPoolExecutor با 3 Thread
with ThreadPoolExecutor(max_workers=3) as executor:
    executor.map(my_task, [1, 2, 3, 4, 5])
```

در اینجا:

- `max_workers=3` به این معناست که حداکثر سه Thread به طور هم‌زمان اجرا می‌شوند.
- `executor.map()` به طور خودکار ورودی‌ها را به Threadها تقسیم می‌کند.

5. کار با مشکلات هم‌زمانی (Concurrency issues)

در برنامه‌هایی که از چندین Thread استفاده می‌کنند، ممکن است مشکلاتی مانند **race condition** پیش آید. این وضعیت زمانی رخ می‌دهد که چندین Thread به داده‌های مشترک دسترسی دارند و تغییرات آنها به طور هم‌زمان ممکن است باعث بروز خطا شود.

5.1. استفاده از `Lock`, `RLock`, `Semaphore`

برای حل این مشکلات، می‌توان از مکانیزم‌های هم‌زمانی مانند `RLock` ، `Lock` و `Semaphore` استفاده کرد.

- **`Lock`**: یک قفل ساده که برای مدیریت دسترسی به منابع مشترک به‌کار می‌رود.
- **`RLock`**: قفل بازگشتی که به یک Thread این امکان را می‌دهد که چندین بار به‌طور متوالی قفل را بگیرد.
- **`Semaphore`**: مشابه `Lock` است اما اجازه می‌دهد تعداد مشخصی از Threadها به‌طور هم‌زمان به منبع دسترسی داشته باشند.

مثال استفاده از `Lock`:

```
import threading

lock = threading.Lock()

def thread_task():
    with lock:
        # در هر زمان قابل اجراست Thread این بخش فقط توسط یک
        print("Thread is working with shared resource.")

# ایجاد چندین Thread
threads = []
for i in range(5):
    t = threading.Thread(target=thread_task)
    threads.append(t)
    t.start()
```

```
# منتظر ماندن برای پایان اجرای همه Threadها
for t in threads:
    t.join()

print("All threads finished.")
```

6. استفاده از `Threading.Event` و `Condition` برای هماهنگی بین Threadها

در بعضی موارد، نیاز به هماهنگی بین Threadها وجود دارد. برای این منظور از `Event` و `Condition` می‌توان استفاده کرد.

- `Event`: این کلاس به شما اجازه می‌دهد تا یک یا چند Thread منتظر یک رویداد خاص باشند.
- `Condition`: برای هماهنگی پیچیده‌تر بین Threadها و مدیریت شرایط خاص.

مثال استفاده از `Event`:

```
import threading
import time

event = threading.Event()

def wait_for_event():
    print("Thread is waiting for event...")
    event.wait()  # منتظر رویداد
    print("Thread resumed after event.")

# ایجاد Thread
t = threading.Thread(target=wait_for_event)
t.start()

# شبیه‌سازی کارهای دیگر
time.sleep(2)

# ایجاد رویداد
print("Event is set!")
event.set()  # ارسال رویداد

t.join()
```

در اینجا:

- `event.wait()` باعث می‌شود که Thread منتظر بماند تا رویداد (`event.set()`) ارسال شود.

7. نتیجه‌گیری

- Threadها واحدهای اجرایی در یک فرآیند هستند که می‌توانند به‌طور هم‌زمان در یک فضای حافظه مشترک اجرا شوند.
- با استفاده از کتابخانه `threading` در پایتون، می‌توانیم Threadها را ایجاد کرده و از ابزارهایی مثل `join`, `start()`, و `is_alive()` برای کنترل اجرای آنها استفاده کنیم.
- برای مدیریت هم‌زمانی و جلوگیری از مشکلاتی مانند `race condition` از مکانیزم‌های `Lock`, `Semaphore`, `RLock`, و `Event` می‌توانیم استفاده کنیم.

- `ThreadPoolExecutor` برای مدیریت مجموعه‌ای از Threadها به‌طور هم‌زمان مفید است.

این مفاهیم به شما کمک می‌کند که برنامه‌های هم‌زمان و کارآمدتری بنویسید.