LAB 5. Thread và MultiThread trong Python (Thời gian thực hiện 60-80 phút)

LAB 5.1: **Chương Trình Đơn Giản với Single Thread và OOP**

Thời gian thực hiện **(**15-20 phút)

Mục Tiêu

- Hiểu cách áp dụng OOP trong Python.
- Hiểu cách một single thread hoạt động trong Python qua việc thực hiện một nhiệm vụ tính toán cụ thể.

Yêu Cầu

- Tạo một class Python với phương thức để thực hiện một nhiệm vụ cụ thể.
- Chương trình khi thực thi sẽ tạo đối tượng từ class này và gọi phương thức tương ứng.

Hướng Dẫn

1. Khởi Tạo Class: Tạo một class (ví dụ: SimpleTask) với một phương thức (ví dụ: run_task) để thực hiện nhiệm vụ.
2. Thực Hiện Nhiệm Vụ: Phương thức run_task có thể in một dãy số hoặc một chuỗi ký tự.
3. Khởi Tạo và Chạy Đối Tượng: Trong hàm main, tạo đối tượng từ class SimpleTask và gọi phương thức run_task.

Code LAB 5.1.

```
1  import time
2
3  class SimpleTask:
4      def run_task(self):
5          for i in range(1, 11):
6              print('Đã in lần thứ :',i)
7              time.sleep(2)  # Dừng 2 giây giữa mỗi lần in
8
9  def main():
10     task = SimpleTask()
11     task.run_task()
12
13 if __name__ == "__main__":
14     main()
```

Kết quả thực hiện chương trình:

```
Đã in lần thứ : 1
Đã in lần thứ : 2
Đã in lần thứ : 3
Đã in lần thứ : 4
Đã in lần thứ : 5
Đã in lần thứ : 6
Đã in lần thứ : 7
Đã in lần thứ : 8
Đã in lần thứ : 9
Đã in lần thứ : 10
```

LAB 5.2: Chuyển Đổi Sang Multi-Threading

Thời gian thực hiện **(**15-20 phút)

Mục Tiêu

- Hiểu rõ hơn về multi-threading.
- So sánh hiệu suất và cách hoạt động giữa single-thread và multi-thread.

Yêu Cầu

Chuyển đổi chương trình từ LAB5.1 để sử dụng nhiều threads.

Hướng Dẫn

1. Chuyển Đổi Class SimpleTask: Bổ sung cách thức để chạy run_task trong một thread riêng biệt.
2. Khởi Tạo Nhiều Threads: Trong hàm main, tạo và khởi chạy nhiều đối tượng SimpleTask, mỗi đối tượng trong một thread riêng.
3. Quan Sát và Phân Tích: So sánh cách thức hoạt động và thời gian hoàn thành giữa phiên bản single-thread và multi-thread.

Code Lab 5.2

```
1  import threading
2  import time
3
4  class SimpleTask:
5      def run_task(self):
6          for i in range(1, 4):
7              print('Đã in lần thứ :',i)
8              time.sleep(1)
9  def main():
10     # Tạo và khởi chạy nhiều threads
11     tasks = [threading.Thread(target=SimpleTask().run_task) for _ in range(4)]
12     for task in tasks:
13         task.start()
14     for task in tasks:
15         task.join()
16
17 if __name__ == "__main__":
18     main()
```

Kết quả thực hiện chương trình

```
Đã in lần thứ : 1
Đã in lần thứ : 1
Đã in lần thứ : 1
Đã in lần thứ : 1
Đã in lần thứ :Đã in lần thứ : 2
Đã in lần thứ : 2
 2
Đã in lần thứ : 2
Đã in lần thứ :Đã in lần thứ : 3
Đã in lần thứ : 3
Đã in lần thứ : 3
 3
```

Trong đoạn code mã lệnh dòng 11:

```
tasks = [threading.Thread(target=SimpleTask().run_task) for _ in range(4)]
```

1. SimpleTask().**run_task**: Tạo một instance mới của class SimpleTask và sau đó gọi phương thức run_task của instance đó.
2. **threading.Thread**(target=**SimpleTask().run_task**): Tạo một thread mới với target là phương thức run_task của một instance SimpleTask.
3. `[for _ in range(4)]`, tạo một list gồm 4 threads, mỗi thread chạy run_task trên một instance riêng biệt của SimpleTask.

Vì vậy, trong đoạn code này, mỗi lần lặp trong vòng lặp for sẽ tạo ra một instance mới của SimpleTask, và mỗi instance này sẽ chạy trong một thread riêng biệt. Điều này được thực hiện 4 lần, tạo ra tổng cộng 4 instances độc lập của SimpleTask, mỗi cái chạy trên một thread riêng.

Kết quả thực hiện chương trình cho thấy: sự hoạt động đồng thời (concurrency) của các threads trong chương trình multi-threading. Sau đây là một số nhận xét quan trọng mà kết quả này đã cho thấy:

1. Thực Hiện Đồng Thời: Các dòng "Đã in lần thứ : 1" từ các threads khác nhau xuất hiện gần như cùng một lúc, cho thấy rằng nhiều threads đang chạy đồng thời.
2. Xuất Ra Không Đồng Bộ: Chúng ta có thể thấy rằng các dòng không được in ra một cách tuần tự hoặc gọn gàng. Thay vào đó, chúng chồng chéo lên nhau. Điều này xảy ra do mỗi thread chạy độc lập và hệ thống điều phối CPU giữa các threads không đảm bảo thứ tự cố định.
3. Chia Sẻ STDOUT: Các threads đang chia sẻ cùng một standard output (STDOUT), tức là màn hình console. Khi nhiều threads cùng ghi dữ liệu vào STDOUT cùng một lúc, dữ liệu đầu ra có thể bị xen kẽ, dẫn đến kết quả đầu ra không được tổ chức một cách rõ ràng.
4. Không Đồng Bộ trong Thời Gian: Mặc dù các threads bắt đầu cùng một lúc, nhưng việc lập lịch thực thi của chúng phụ thuộc vào hệ điều hành và có thể thay đổi, dẫn đến việc in ra các giá trị không theo một trình tự tuần tự.

Kết Luận và Ý Nghĩa

Kết quả này cho thấy tính chất của multi-threading, nơi mỗi thread hoạt động độc lập và có thể giao tiếp hoặc tương tác với các resources chung (như STDOUT ở đây) một cách không đồng bộ. Điều này rất quan trọng trong việc hiểu và xử lý các vấn đề liên quan đến multi-threading như race conditions, đồng bộ hóa, và quản lý tài nguyên chung.

LAB 3. Đồng bộ hóa và chia sẻ tài nguyên.

LAB 5.3.1. Hiện Tượng Race Condition1.

Thời gian thực hiện **(**15-20 phút)

Mục Tiêu

- Hiểu về hiện tượng race condition trong môi trường multi-threading.
- Quan sát cách thức các threads tương tác với shared resource khi không có sự quản lý.

Yêu Cầu: Chỉnh sửa chương trình từ Lab 5.2 để các threads cùng thay đổi một **shared resource** mà không sử dụng **lock.**

Hướng Dẫn:

Chỉnh Sửa Phương Thức run_task: Mỗi thread sẽ tăng giá trị của shared resource (ví dụ: biến counter) mà không sử dụng lock.

Chạy và Quan Sát: Chạy chương trình và quan sát cách counter được tăng lên. Lưu ý xem liệu giá trị cuối cùng có phản ánh đúng số lần tăng hay không.

Code Lab 5.3.1.py

```
1   class SimpleTask:
2       def run_task(self):
3           global counter
4           for _ in range(4):
5               time.sleep(2)
6               counter += 1  # Tăng counter mà không sử dụng lock
7               print(f"Counter đã tăng lên: {counter}")
8
9   def main():
10      tasks = [threading.Thread(target=SimpleTask().run_task) for _ in range(4)]
11      for task in tasks:
12          task.start()
13      for task in tasks:
14          task.join()
15
16  if __name__ == "__main__":
17      main()
```

Kết quả thực hiện chương trình

```
Counter đã tăng lên: 1Counter đã tăng lên: 2
Counter đã tăng lên: 3
Counter đã tăng lên: 4

Counter đã tăng lên: 5Counter đã tăng lên: 6
Counter đã tăng lên: 7

Counter đã tăng lên: 8
Counter đã tăng lên: 9
Counter đã tăng lên: 10
Counter đã tăng lên: 11
Counter đã tăng lên: 12
Counter đã tăng lên: 13Counter đã tăng lên: 14

Counter đã tăng lên: 15
Counter đã tăng lên: 16
```

Nhận xét và bình luận:

Một số điểm quan trọng liên quan đến cách hoạt động của multi-threading và vấn đề race condition:

1. **Số Lần Tăng counter Phản Ánh Số Lần Lặp và Số Threads**: Với mỗi thread thực hiện vòng lặp 4 lần và tổng cộng 3 threads, counter được tăng tổng cộng 12 lần (4 lần mỗi thread x 3 threads = 12). Kết quả cuối cùng là counter tăng đến 12, phù hợp với kỳ vọng này.
2. **Xảy Ra Race Condition**: Mặc dù kết quả cuối cùng ("Counter đã tăng lên: 16") là chính xác, nhưng quá trình tăng giá trị counter vẫn chịu ảnh hưởng của race condition. Các threads cùng cố gắng truy cập và thay đổi counter mà không có sự đồng bộ hóa, dẫn đến việc không thể đảm bảo rằng mỗi lần tăng counter sẽ được thực hiện một cách riêng biệt và an toàn.

3. **Xuất Ra Chồng Chéo**: Kết quả đầu ra ("Counter đã tăng lên: ...") vẫn xuất hiện một cách không đồng nhất, phản ánh sự không đồng bộ trong cách các threads giao tiếp với STDOUT.

LAB 5.3.2. Xử Lý Race Condition Bằng Locks

Thời gian thực hiện : (15-20 phút).

Mục Tiêu

- Hiểu cách sử dụng locks để ngăn chặn race condition trong môi trường multi-threading.
- Quan sát sự thay đổi trong kết quả khi sử dụng locks.

Yêu Cầu

- Chỉnh sửa chương trình từ Lab 3.1 để sử dụng locks khi các threads truy cập và thay đổi shared resource.

Hướng Dẫn

1. **Thêm Lock**: Tạo một đối tượng Lock bên ngoài class SimpleTask.
2. Sử dụng Lock trong phương thức **run_task**(): Chỉnh sửa phương thức để sử dụng lock khi tăng giá trị của counter.
3. Chạy và phân Tích: Chạy chương trình đã sửa và so sánh kết quả với LAB 5. 3.1.

Code Lab 5.3.2.py

```
1  import threading
2  import time
3
4  counter = 0
5  counter_lock = threading.Lock()
6
7  class SimpleTask:
8      def run_task(self):
9          global counter
10         for _ in range(3):
11             time.sleep(2)
12             with counter_lock:
13                 counter += 1
14                 print(f"Counter đã tăng lên: {counter}")
15
16 def main():
17     tasks = [threading.Thread(target=SimpleTask().run_task) for _ in range(4)]
18     for task in tasks:
19         task.start()
20     for task in tasks:
21         task.join()
22
23 if __name__ == "__main__":
24     main()
```

Kết quả thực hiện chương trình.

```
Counter đã tăng lên: 1
Counter đã tăng lên: 2
Counter đã tăng lên: 3
```

```
Counter đã tăng lên: 4
Counter đã tăng lên: 5
Counter đã tăng lên: 6
Counter đã tăng lên: 7
Counter đã tăng lên: 8
Counter đã tăng lên: 9
Counter đã tăng lên: 10
Counter đã tăng lên: 11
Counter đã tăng lên: 12
```

Nhận xét: locks giúp đảm bảo rằng chỉ một thread có thể thay đổi counter tại một thời điểm, ngăn chặn race condition.