1. **Nêu 1 bài toán sử dụng đa luồng liên quan data**

Mô tả bài toán:

Giả sử bạn có một tập dữ liệu lớn gồm các văn bản cần được phân loại thành các chủ đề khác nhau (ví dụ: thể thao, chính trị, kinh doanh, v.v.). Việc phân loại văn bản truyền thống sẽ thực hiện tuần tự từng văn bản, dẫn đến thời gian xử lý lâu khi tập dữ liệu lớn.

Giải pháp đa luồng:

Để tăng tốc độ phân loại, ta có thể sử dụng đa luồng bằng cách chia tập dữ liệu thành nhiều phần nhỏ và phân loại mỗi phần trên một luồng riêng biệt. Sau khi phân loại xong, kết quả từ các luồng sẽ được tổng hợp lại để có kết quả cuối cùng.

2. Cơ chế đa luồng của python

Các cơ chế đồng bộ trong lập trình đa luồng của Python:

1. Lock

Lock là cơ chế đồng bộ cơ bản nhất của Python. Một Lock gồm có 2 trạng thái, locked và unlocked, cùng với 2 phương thức để thao tác với Lock là acquire() và release(). Các quy luật trên Lock là:

- Nếu trạng thái (state) là unlocked, gọi acquire() sẽ thay đổi trạng thái sang locked
- Nếu trạng thái là locked, tiến trình gọi acquire() sẽ phải đợi (block) cho đến khi tiến trình khác gọi method release().
- Nếu trạng thái là unlocked, gọi method release() sẽ phát ra RuntimeError exception.
- Nếu trạng thái là locked, gọi method release() sẽ chuyển trạng thái của Lock sang unlocked.

2. Rlock (Reentrant Lock)

Tương tự như Lock và sử dụng khái niệm "owning thread" (tiến trình sở hữu) và "recursion level" (có thể gọi method acquire() nhiều lần).

- Owning thread: Rlock object lưu giữ định danh (ID) của thread sở hữu nó. Do đó, khi một thread sở hữu Rlock, thread đó có thể gọi method acquire() nhiều lần mà không bị block (Khác với Lock, khi một thread đã dành được khóa, các tiến trình gọi acquire() sẽ bị block cho đến khi khóa được release). Hơn nữa, đối với Rlock, khóa chỉ được unlocked khi thread sở hữu gọi method release() (các lời gọi acquire() và release() có thể lồng nhau) và số lần gọi release() phải bằng với số lần gọi method acquire() trước đó (khác với Lock, khóa có thể được unlocked khi một thread bất kỳ gọi method release()).
- recursion level: Một khi thread dành được khóa, thread đó có thể gọi acquire() nhiều lần mà không bị block.

3. Condition

Đây là cơ chế đồng bộ được sử dụng khi một thread muốn đợi (block) một điều kiện nào đấy xảy ra và một thread khác sẽ thông báo (notify) khi điều kiện đã xảy ra, lúc đấy thread đang đợi sẽ được đánh thức và dành được khóa để truy cập độc quyền vào tài nguyên chung (không có thread nào khác được phép truy cập vào tài nguyên khi thread này chưa realse khóa):

- Biến điều kiện luôn liên kết với một khóa bên dưới (Lock hoặc RLock). Biến điều kiện cung cấp các method chính như sau:
- acquire(*args): Method này đơn giản chỉ trả về method acquire() của khóa bên dưới.
- release(): Tương tự acquire(), gọi mthod release() của khóa bên dưới.
- wait(timeout=None): Đợi cho đến khi được thông báo (notify) hoặc timeout. Nếu thread gọi method này khi chưa dành đươc khóa thì một RuntimeError sẽ được ném ra. Phương thức này giải phóng khóa bên dưới và block cho đến khi một thread khác gọi notify() hay notify_all() trên cùng biến điều kiện, hoặc timeout xảy ra. Một khi được thức giấc, thread dành lại khóa của mình và tiếp tục thực hiện công việc.
- notify(n=1): Method dùng để đánh thức (notify) nhiều nhất là n thread đang đợi điều kiện này xảy ra. RuntimeError exception sẽ được ném ra nếu thread gọi hàm này vẫn chưa dành (acquire) được khóa bên dưới. Lưu ý rằng method notify() không giải phóng khóa bên dưới, do đó thread gọi notify() cần phải gọi release() method tường minh để các thread đang đợi điều kiện (wait()) có thể dành được khóa.
- notify_all(): Đánh thức tất cả các thread đang đợi điều kiện này. Method này chỉ đơn giản gọi notify() với đối số n là tất cả các thread.

4. Semaphore:
    - Đây là một trong những cơ chế đồng bộ lâu đời nhất trong lịch sử khoa học máy tính, được phát minh bởi nhà khoa học máy tính Edsger W. Dijkstra (trong đó ông sử dụng P() và V() thay vì acquire() và release()).
    - Một Semaphore duy trì một biến đếm (không âm) được truyền vào khi khởi tạo một Semaphore object, có giá trị giảm sau mỗi lần gọi acquire()

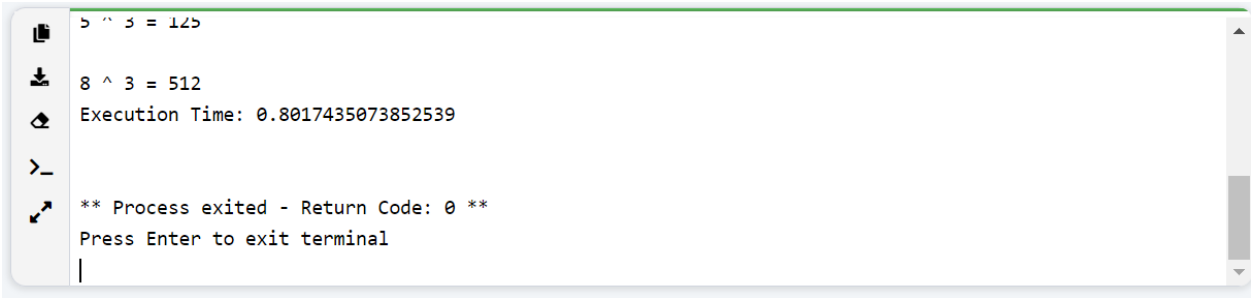
và tăng sau mõi lần gọi method release(). Khi gọi method acquire() trên một Semaphore object với giá trị biến đếm bằng 0, nó sẽ block thread gọi và đợi cho đến khi thread khác gọi release() (làm tăng giá trị biến đếm lên 1).

- Semaphore cũng cung cấp 2 method là acquire() và release() tương tự như cơ chế Lock hay RLock, chỉ khác ở chỗ method acquire() sẽ trả về ngay tức thì khi biến đếm có giá trị lớn hơn không.
- Với cơ chế như trên, Semaphore thường được dùng để giới hạn số lượng thread đồng thời truy cập vào tài nguyên chúng, ví dụ, giới hạn số lượng kết nối tới một database server. Dễ nhận thấy rằng Lock là một trường hợp riêng của Semaphore trong đó biến đếm được khởi tạo bằng 1.

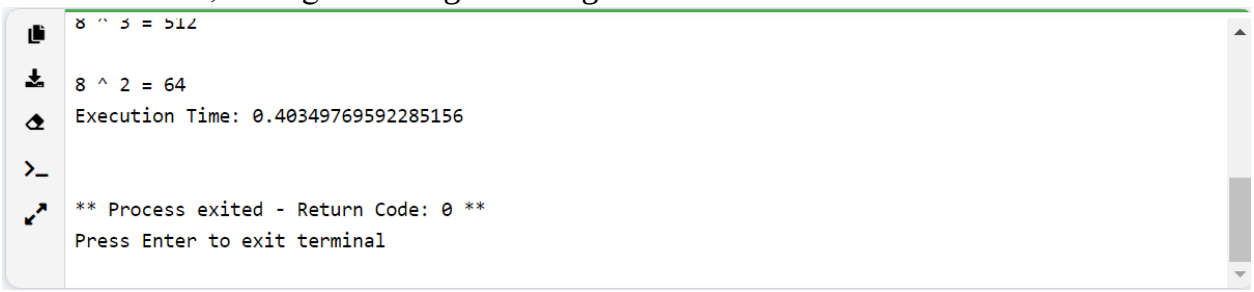
5. Event:
    - Event là cơ chế đồng bộ đơn giản trong đó một thread sẽ phát ra một sự kiện (event) và các thread khác đợi sự kiện đó.
    - Một Event object quản lý một cờ trong (internal flag), được thiết lập True với method set() và thiết lập False với clear(). Tiến trình gọi method wait() bị block cho đến khi cờ này có giá trị True.
    - Quay trở lại với bài toán producer/consumer ở trên, ta sẽ cài đặt nó sử dụng Event thay vì Condition. Mỗi lần một số nguyên được thêm vào mảng chung, sự kiện sẽ được thiết lập (set) và cờ sẽ được clear ngay sau đó để thông báo tới consumer. Lưu ý rằng cờ được clear (False) khi khởi tạo Event object.

3. Code bài toán đa luồng đơn giản trong python:

Với code hiện tại ta có thời gian thực thi là 0.8, tức là khi mà CPU idle khi tính toán cpu



Với code thứ 2, ta đã giảm thời gian xuống còn




```python
# """ Thư viện cho biết thời gian thực thi của chương trình """
import time

# """ Thư viện hỗ trợ multithreading """
import threading

# """ Hàm tính diện tích hình vuông """
def calc_square(numbers):
    for n in numbers:
        print(f'\n{n} ^ 2 = {n*n}')
        time.sleep(0.1)
        # cho ngủ 0.1 giây

# """ Hàm tính thể tích hình lập phương """
def calc_cube(numbers):
    for n in numbers:
        print(f'\n{n} ^ 3 = {n*n*n}')
        time.sleep(0.1)

numbers = [2, 3, 5, 8]
start = time.time()

# """ Thread 1: xử lý hàm calc_square"""
square_thread = threading.Thread(target=calc_square, args=(numbers,))
```

```python
# """ Thread 2 xử lý hàm calc_cube"""
cube_thread = threading.Thread(target=calc_cube, args=(numbers,))

# """ Gọi thread """
square_thread.start()
cube_thread.start()

square_thread.join()
cube_thread.join()

end = time.time()

print('Execution Time: {}'.format(end-start))
```