BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

TRƯỜNG CNTT & TT



BÁO CÁO BTL MÔN NGUYÊN LÝ HỆ ĐIỀU HÀNH – IT3070

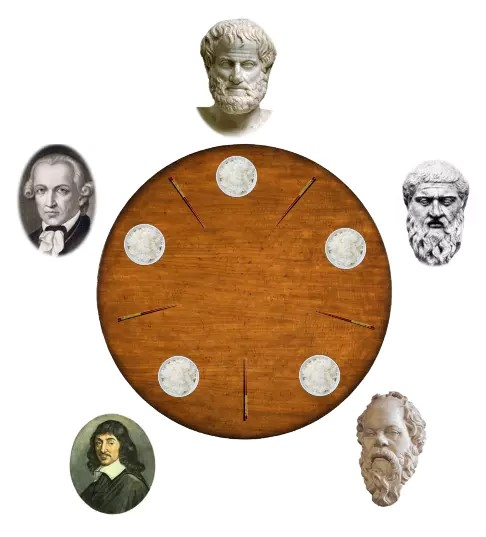
CHỦ ĐỀ : Xây dựng chương trình minh họa cho giải pháp bài toán ***Bữa tối của triết gia***

NGƯỜI THỰC HIỆN: Phạm Tấn Hưng – MSSV 20215063

MÃ LỚP HỌC: 141310

1. **TỔNG QUAN**

* Bài toán ***Bữa tối của triết gia***
* Có năm nhà triết gia, vừa suy nghĩ vừa ăn tối. Các triết gia ngồi trên một bàn tròn, trước mặt họ là các đĩa thức ăn, mỗi người một đĩa. Có 5 chiếc đũa được đặt xen kẽ giữa các triết gia. Như hình vẽ:



* Bài toán được phát biểu như sau: "Khi một triết gia suy nghĩ, ông ta không giao tiếp với các triết gia khác. Thỉnh thoảng, một triết gia cảm thấy đói và cố gắng chọn hai chiếc đũa gần nhất (hai chiếc đũa nằm giữa ông ta với hai láng giềng trái và phải). Một triết gia có thể lấy chỉ một chiếc đũa tại một thời điểm. Chú ý, ông ta không thể lấy chiếc đũa mà nó đang được dùng bởi người láng giềng. Khi một triết gia đói và có hai chiếc đũa cùng một lúc, ông ta ăn mà không đặt đũa xuống. Khi triết gia ăn xong, ông ta đặt đũa xuống và bắt đầu suy nghĩ tiếp."
* **Tìm cách để không ai phải chết đói là vấn đề của bài toán.**
* Ý nghĩa bài toán và Deadlock:
* Đây là bài toán kinh điển về đồng bộ hóa. Nó trình bày quá trình cấp phát nhiều tài nguyên giữa các tiến trình mà không bị khoá chết và đói tài nguyên.
* Deadlock: Là trạng thái hai hay nhiều tiến trình cùng chờ đợi một sự kiện nào đó xảy ra. Nếu không có tác động gì từ bên ngoài thì sự chờ đợi đó là vô hạn.
* Điều kiện xảy ra Deadlock:

+ Tồn tại tài nguyên găng (Mutual Exclusion)

+ Chờ đợi trước khi vào đoạn găng (Hold and Wait)

+ Không có hệ thống phân phối lại tài nguyên (No Preemption)

+ Chờ đợi vòng tròn (Circular Wait)

Deadlock chỉ xảy ra khi có đủ cả 4 điều kiện trên.

1. **GIẢI QUYẾT BÀI TOÁN**
   * + 1. **Công cụ hỗ trợ:**

* Ngôn ngữ lập trình Python
* VS Code và Cmd trên Window 11
  + - 1. **Cách 1:** sử dụng khóa (locks) để đồng bộ hóa truy cập đến các tài nguyên chung như đôi đũa. Mỗi triết gia sử dụng khóa để kiểm tra trạng thái của đôi đũa và lấy đũa khi cả hai đều có sẵn.
* **Thuật toán:**
* Trong chương trình, mỗi triết gia được đại diện bởi một luồng (thread). Mỗi triết gia được thực hiện trong phương thức **philosopher**. Các triết gia thực hiện việc suy nghĩ, lấy đũa và ăn món ăn trong một vòng lặp cho đến khi hết chén món ăn của mình.
* Trong mỗi lần lặp, triết gia đầu tiên thử lấy đũa bên trái, sau đó thử lấy đũa bên phải. Nếu cả hai đũa đều có sẵn, triết gia đó được cho là đang ăn (trạng thái "E"). Sau khi ăn xong, triết gia đặt lại hai đũa và chuyển trạng thái về "T" (đang suy nghĩ). Nếu một trong hai đũa không có sẵn, triết gia đặt lại đũa đã lấy và tiếp tục suy nghĩ.
* Trong quá trình chạy chương trình, trạng thái của mỗi triết gia và trạng thái các đôi đũa được in ra màn hình sau mỗi khoảng thời gian ngắn. Khi tất cả chén món ăn đã được tiêu thụ hết, chương trình kết thúc.
* **Mã nguồn:**

from threading import Thread, Lock

import random

import time

class DiningPhilosophers:

    def \_\_init\_\_(self, number\_of\_philosophers, meal\_size=9):

*self*.meals = [meal\_size for \_ in range(number\_of\_philosophers)]

*self*.chopsticks = [Lock() for \_ in range(number\_of\_philosophers)]

*self*.status = ["  T  " for \_ in range(number\_of\_philosophers)]

*self*.chopstick\_holders = ["     " for \_ in range(number\_of\_philosophers)]

*self*.number\_of\_philosophers = number\_of\_philosophers

    def philosopher(self, i):

        j = (i + 1) % *self*.number\_of\_philosophers

        while *self*.meals[i] > 0:

*self*.status[i] = "  T  "

            time.sleep(random.random())

*self*.status[i] = "  \_  "

            if not *self*.chopsticks[i].locked():

*self*.chopsticks[i].acquire()

*self*.chopstick\_holders[i] = " /   "

                time.sleep(random.random())

                if not *self*.chopsticks[j].locked():

*self*.chopsticks[j].acquire()

*self*.chopstick\_holders[i] = " / \\ "

*self*.status[i] = "  E  "

                    time.sleep(random.random())

*self*.meals[i] -= 1

*self*.chopsticks[j].release()

*self*.chopstick\_holders[i] = "     "

*self*.chopsticks[i].release()

*self*.chopstick\_holders[i] = "     "

*self*.status[i] = "  T  "

                else:

*self*.chopsticks[i].release()

*self*.chopstick\_holders[i] = "     "

def main():

    n = 5

    m = 7

    dining\_philosophers = DiningPhilosophers(n, m)

    philosophers = [

        Thread(target=dining\_philosophers.philosopher, args=(i,)) for i in range(n)

    ]

    for philosopher in philosophers:

        philosopher.start()

    while sum(dining\_philosophers.meals) > 0:

        print("=" \* (n \* 5))

        print(

            "".join(map(str, dining\_philosophers.status)),

            " : ",

            str(dining\_philosophers.status.count("  E  ")),

        )

        print("".join(map(str, dining\_philosophers.chopstick\_holders)))

        print(

            "".join("{:3d}  ".format(m) for m in dining\_philosophers.meals),

            " : ",

            str(sum(dining\_philosophers.meals)),

        )

        time.sleep(0.1)

    for philosopher in philosophers:

        philosopher.join()

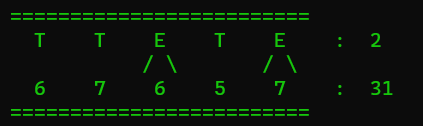
if \_\_name\_\_ == "\_\_main\_\_":

    main()

* + - * **Phân tích mã nguồn và thuật toán sử dụng:**
* **from threading import Thread, Lock**: Import các module Thread và Lock từ thư viện threading. Module Thread được sử dụng để tạo và quản lý các luồng (threads) trong Python. Module Lock được sử dụng để tạo khóa (lock) đồng bộ hóa trong các tác vụ đa luồng.
* **import random**: Import module random từ thư viện random. Module này cung cấp các công cụ liên quan đến số ngẫu nhiên.
* **import time**: Import module time từ thư viện time. Module này cung cấp các công cụ liên quan đến thời gian.
* **class DiningPhilosophers:**: Định nghĩa một lớp (class) có tên là DiningPhilosophers.
* **def \_\_init\_\_(self, number\_of\_philosophers, meal\_size=9):**: Định nghĩa phương thức khởi tạo (constructor) cho lớp DiningPhilosophers, nhận vào tham số number\_of\_philosophers (số lượng triết gia) và meal\_size (kích cỡ bữa ăn, mặc định là 9).
* **self.meals = [meal\_size for \_ in range(number\_of\_philosophers)]**: Tạo một danh sách meals gồm number\_of\_philosophers phần tử, mỗi phần tử có giá trị là meal\_size. Danh sách này đại diện cho số lượng bữa ăn của mỗi triết gia.
* **self.chopsticks = [Lock() for \_ in range(number\_of\_philosophers)]**: Tạo một danh sách chopsticks gồm number\_of\_philosophers khóa Lock. Danh sách này đại diện cho các đũa ăn của mỗi triết gia.
* **self.status = [" T " for \_ in range(number\_of\_philosophers)]**: Tạo một danh sách status gồm number\_of\_philosophers phần tử, mỗi phần tử có giá trị là chuỗi " T ". Danh sách này đại diện cho trạng thái của mỗi triết gia.
* **self.chopstick\_holders = [" " for \_ in range(number\_of\_philosophers)]**: Tạo một danh sách chopstick\_holders gồm number\_of\_philosophers chuỗi trống. Danh sách này đại diện cho trạng thái của các đũa ăn.
* **self.number\_of\_philosophers = number\_of\_philosophers**: Gán giá trị number\_of\_philosophers cho thuộc tính number\_of\_philosophers của đối tượng DiningPhilosophers.
* **def philosopher(self, i):**: Định nghĩa phương thức philosopher cho lớp DiningPhilosophers, nhận vào tham số i (chỉ số của triết gia).
* **j = (i + 1) % self.number\_of\_philosophers**: Tính chỉ số của đũa ăn bên phải của triết gia hiện tại bằng cách lấy phần dư của (i + 1) chia cho self.number\_of\_philosophers.
* **while self.meals[i] > 0:**: Lặp lại cho đến khi số lượng bữa ăn của triết gia i lớn hơn 0.
* **self.status[i] = " T "**: Gán giá trị " T " cho phần tử thứ i trong danh sách status, đại diện cho triết gia i đang suy nghĩ.
* **time.sleep(random.random())**: Dừng một khoảng thời gian ngẫu nhiên.
* **self.status[i] = " \_ "**: Gán giá trị " \_ " cho phần tử thứ i trong danh sách status, đại diện cho triết gia i đã kết thúc suy nghĩ và sẽ ăn.
* **if not self.chopsticks[i].locked():**: Kiểm tra xem đũa ăn của triết gia i có được khóa hay không.
* **self.chopsticks[i].acquire()**: Khóa đũa ăn của triết gia i.
* **self.chopstick\_holders[i] = " / "**: Gán giá trị " / " cho phần tử thứ i trong danh sách chopstick\_holders, đại diện cho triết gia i đã lấy đũa ăn bên trái.
* **time.sleep(random.random())**: Dừng một khoảng thời gian ngẫu nhiên.
* **if not self.chopsticks[j].locked():**: Kiểm tra xem đũa ăn bên phải của triết gia i có được khóa hay không.
* **self.chopsticks[j].acquire()**: Khóa đũa ăn bên phải của triết gia i.
* **self.chopstick\_holders[i] = " / \\ "**: Gán giá trị " / \ " cho phần tử thứ i trong danh sách chopstick\_holders, đại diện cho triết gia i đã lấy cả hai đũa ăn.
* **self.status[i] = " E "**: Gán giá trị " E " cho phần tử thứ i trong danh sách status, đại diện cho triết gia i đang ăn.
* **time.sleep(random.random())**: Dừng một khoảng thời gian ngẫu nhiên.
* **self.meals[i] -= 1**: Giảm số lượng bữa ăn của triết gia i đi 1.
* **self.chopsticks[j].release()**: Giải phóng đũa ăn bên phải của triết gia i.
* **self.chopstick\_holders[i] = " "**: Gán giá trị " " cho phần tử thứ i trong danh sách chopstick\_holders, đại diện cho triết gia i đã giải phóng đũa ăn.
* **self.chopsticks[i].release()**: Giải phóng đũa ăn của triết gia i.
* **self.chopstick\_holders[i] = " "**: Gán giá trị " " cho phần tử thứ i trong danh sách chopstick\_holders, đại diện cho triết gia i đã giải phóng đũa ăn.
* **self.status[i] = " T "**: Gán giá trị " T " cho phần tử thứ i trong danh sách status, đại diện cho triết gia i đã kết thúc ăn và quay trở lại suy nghĩ.
* **def main():**: Định nghĩa phương thức main, là điểm bắt đầu của chương trình.
* **n = 5**: Gán giá trị 5 cho biến n, đại diện cho số lượng triết gia.
* **m = 7**: Gán giá trị 7 cho biến m, đại diện cho kích cỡ bữa ăn.
* **dining\_philosophers = DiningPhilosophers(n, m)**: Tạo một đối tượng dining\_philosophers từ lớp DiningPhilosophers với số lượng triết gia n và kích cỡ bữa ăn m.
* **philosophers = [Thread(target=dining\_philosophers.philosopher, args=(i,)) for i in range(n)]**: Tạo một danh sách philosophers gồm n luồng (threads), mỗi luồng thực thi phương thức philosopher của đối tượng dining\_philosophers với đối số là chỉ số của triết gia tương ứng.
* **for philosopher in philosophers: philosopher.start()**: Khởi chạy tất cả các luồng trong danh sách philosophers.
* **while sum(dining\_philosophers.meals) > 0: ...**: Lặp lại cho đến khi tổng số lượng bữa ăn của tất cả triết gia lớn hơn 0.
* **print("=" \* (n \* 5))**: In ra dấu "=" lặp lại n lần 5 lần để tạo đường kẻ ngăn cách.
* **print("".join(map(str, dining\_philosophers.status)), " : ", str(dining\_philosophers.status.count(" E ")))**: In ra danh sách trạng thái của các triết gia, số triết gia đang ăn và số lượng bữa ăn còn lại.
* **print("".join(map(str, dining\_philosophers.chopstick\_holders)))**: In ra danh sách trạng thái của các đũa ăn.
* **print("".join("{:3d} ".format(m) for m in dining\_philosophers.meals), " : ", str(sum(dining\_philosophers.meals)))**: In ra danh sách số lượng bữa ăn của các triết gia và tổng số lượng bữa ăn còn lại.
* **time.sleep(0.1)**: Dừng một khoảng thời gian 0.1 giây.
* **for philosopher in philosophers: philosopher.join()**: Chờ cho tất cả các luồng trong danh sách philosophers kết thúc thực thi.
* **if \_\_name\_\_ == "\_\_main\_\_":**: Kiểm tra xem module hiện tại có phải là module chính hay không.
* **main()**: Gọi phương thức main để bắt đầu chương trình.
  + - * **Kết quả chạy chương trình**:
* Video:



* Giải thích kết quả: Giả sử kết quả chạy đến bước sau:



1. **=========================**: Đường kẻ ngăn cách giữa các bước thực hiện trong quá trình chạy chương trình.
2. **T T E T E**: Trạng thái của các triết gia. Ký tự "T" đại diện cho triết gia đang suy nghĩ (thinking), và ký tự "E" đại diện cho triết gia đang ăn (eating). Trong trường hợp này, có 2 triết gia đang ăn.
3. **: 2**: Số lượng triết gia đang ăn. Trong trường hợp này, có 2 triết gia đang ăn.
4. **/ \ / \**: Trạng thái của các đũa ăn. Ký tự "/" đại diện cho đũa ăn đang được sử dụng bởi một triết gia, và ký tự " " đại diện cho đũa ăn không được sử dụng. Trong trường hợp này, có 2 cặp đũa ăn đang được sử dụng bởi 2 triết gia.
5. **6 7 6 5 7**: Số lượng bữa ăn còn lại của mỗi triết gia. Mỗi số tương ứng với một triết gia. Trong trường hợp này, số lượng bữa ăn còn lại lần lượt là 6, 7, 6, 5, 7.
6. **: 31**: Tổng số lượng bữa ăn còn lại của tất cả các triết gia. Trong trường hợp này, tổng số lượng bữa ăn còn lại là 31.
   * + 1. **Cách 2: Đèn báo – Semaphore**

* Thuật toán Semaphore trong giải quyết bài toán ***Bữa tối của triết gia***: Semaphore là lời giải kinh điển của bài toán Bữa tối của triết gia, mặc dù nó không ngăn được hết các deadlock. Cụ thể bài toán được trình bày thông qua mã giả:

+ Gọi Semaphore s: Initial s > 0  
+ Chờ đợi (waiting) list được gọi f(s): Initial: f(s) = null, {f(s): sử dụng FIFO nếu có nhiều Ps}

+ P, Q là biến con trỏ, trỏ đến một tiến trình

+ Hai phép toán Down(s) & Up(s): Không thể bị gián đoạn trong quá trình thực hiện, với:

* Down(s): block 1 tiến trình, put into f(s)

void Down(int s){

    s = s - 1;

    if (s < 0)

    {

        status(P) = block;

        enter(P.f(s));

    }

}

* Up(s): wake up 1 tiến trình từ f(s):

void Up(int s){

    s = s + 1;

    if (s <= 0)

    {

        exit(Q.f(s));

        status(Q) = ready;

        enter(Q.ready-list);

    }

}

* **Mã nguồn:**

import sys

import threading

import time

class Semaphore(object):

    def \_\_init\_\_(self, initial):

*self*.lock = threading.Condition(threading.Lock())

*self*.value = initial

    def up(self):

        with *self*.lock:

*self*.value += 1

*self*.lock.notify()

    def down(self):

        with *self*.lock:

            while *self*.value == 0:

*self*.lock.wait()

*self*.value -= 1

class ChopStick(object):

    def \_\_init\_\_(self, number):

*self*.number = number           *# chop stick ID*

*self*.user = -1                 *# keep track of philosopher using it*

*self*.lock = threading.Condition(threading.Lock())

*self*.taken = False

    def take(self, user):         *# used for synchronization*

        with *self*.lock:

            while *self*.taken == True:

*self*.lock.wait()

*self*.user = user

*self*.taken = True

            sys.stdout.write("p[%s] took c[%s]\n" % (user, *self*.number))

*self*.lock.notifyAll()

    def drop(self, user):         *# used for synchronization*

        with *self*.lock:

            while *self*.taken == False:

*self*.lock.wait()

*self*.user = -1

*self*.taken = False

            sys.stdout.write("p[%s] dropped c[%s]\n" % (user, *self*.number))

*self*.lock.notifyAll()

class Philosopher (threading.Thread):

    def \_\_init\_\_(self, number, left, right, butler):

        threading.Thread.\_\_init\_\_(*self*)

*self*.number = number            *# philosopher number*

*self*.left = left

*self*.right = right

*self*.butler = butler

    def run(self):

        for i in range(20):

*self*.butler.down()              *# start service by butler*

            time.sleep(0.1)                 *# think*

*self*.left.take(*self*.number)     *# pickup left chopstick*

            time.sleep(0.1)                 *# (yield makes deadlock more likely)*

*self*.right.take(*self*.number)    *# pickup right chopstick*

            time.sleep(0.1)                 *# eat*

*self*.right.drop(*self*.number)    *# drop right chopstick*

*self*.left.drop(*self*.number)     *# drop left chopstick*

*self*.butler.up()                *# end service by butler*

        sys.stdout.write("p[%s] finished thinking and eating\n" % *self*.number)

def main():

*# number of philosophers / chop sticks*

    n = 5

*# butler for deadlock avoidance (n-1 available)*

    butler = Semaphore(n-1)

*# list of chopsticks*

    c = [ChopStick(i) for i in range(n)]

*# list of philsophers*

    p = [Philosopher(i, c[i], c[(i+1)%n], butler) for i in range(n)]

    for i in range(n):

        p[i].start()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

* + - * **Phân tích mã nguồn:**
* import module sys để sử dụng hàm **stdout** để in ra màn hình, import module threading để sử dụng luồng (thread) và đồng bộ hóa, import module time để sử dụng hàm sleep để tạm dừng thực thi trong một khoảng thời gian nhất định:
* Lớp **Semaphore** được định nghĩa để quản lý Semaphore và các phương thức liên quan. Phương thức **\_\_init\_\_** được sử dụng để khởi tạo Semaphore với giá trị ban đầu và sử dụng một đối tượng khóa (**Lock**) để đồng bộ hóa.Phương thức **up** được sử dụng để tăng giá trị của Semaphore lên 1 và thông báo cho bất kỳ luồng nào đang chờ đợi. Phương thức **down** được sử dụng để giảm giá trị của Semaphore xuống 1 và chờ đến khi giá trị khác 0:

import sys

import threading

import time

class Semaphore(object):

    def \_\_init\_\_(self, initial):

*self*.lock = threading.Condition(threading.Lock())

*self*.value = initial

    def up(self):

        with *self*.lock:

*self*.value += 1

*self*.lock.notify()

    def down(self):

        with *self*.lock:

            while *self*.value == 0:

*self*.lock.wait()

*self*.value -= 1

* Lớp **ChopStick** được định nghĩa để quản lý đôi đũa và các phương thức liên quan. Mỗi đôi đũa có một số hiệu (number) và một biến user để theo dõi triết gia đang sử dụng. Phương thức **take** được sử dụng để lấy đôi đũa. Nếu đôi đũa đã được lấy, triết gia phải chờ đến khi nó được thông báo bởi phương thức **notifyAll**. Khi lấy được đôi đũa, triết gia sẽ cập nhật user và taken, sau đó in ra thông báo sử dụng **sys.stdout.write**. Phương thức **drop** được sử dụng để trả lại đôi đũa. Nếu đôi đũa chưa được lấy, triết gia phải chờ đến khi nó được thông báo bởi phương thức **notifyAll**. Khi trả lại đôi đũa, triết gia sẽ cập nhật user và taken, sau đó in ra thông báo sử dụng **sys.stdout.write**:

class ChopStick(object):

    def \_\_init\_\_(self, number):

*self*.number = number           *# chop stick ID*

*self*.user = -1                 *# keep track of philosopher using it*

*self*.lock = threading.Condition(threading.Lock())

*self*.taken = False

    def take(self, user):         *# used for synchronization*

        with *self*.lock:

            while *self*.taken == True:

*self*.lock.wait()

*self*.user = user

*self*.taken = True

            sys.stdout.write("p[%s] took c[%s]\n" % (user, *self*.number))

*self*.lock.notifyAll()

    def drop(self, user):         *# used for synchronization*

        with *self*.lock:

            while *self*.taken == False:

*self*.lock.wait()

*self*.user = -1

*self*.taken = False

            sys.stdout.write("p[%s] dropped c[%s]\n" % (user, *self*.number))

*self*.lock.notifyAll()

* Lớp **Philosopher** được định nghĩa để tạo các triết gia là các luồng riêng biệt. Phương thức **run** được gọi khi triết gia được bắt đầu thực thi. Trong vòng lặp **for**, triết gia sẽ thực hiện các hành động theo trình tự: triết gia giảm giá trị Semaphore (**butler.down()**), suy nghĩ trong 0.1 giây (**time.sleep(0.1)**), lấy đôi đũa trái (**self.left.take(self.number)**), suy nghĩ trong 0.1 giây, lấy đôi đũa phải (**self.right.take(self.number)**), suy nghĩ trong 0.1 giây, trả lại đôi đũa phải (**self.right.drop(self.number)**), trả lại đôi đũa trái (**self.left.drop(self.number)**), và tăng giá trị Semaphore (**butler.up()**). Sau khi vòng lặp kết thúc, triết gia sẽ in ra thông báo sử dụng **sys.stdout.write**:

class Philosopher (threading.Thread):

    def \_\_init\_\_(self, number, left, right, butler):

        threading.Thread.\_\_init\_\_(*self*)

*self*.number = number            *# philosopher number*

*self*.left = left

*self*.right = right

*self*.butler = butler

    def run(self):

        for i in range(20):

*self*.butler.down()              *# start service by butler*

            time.sleep(0.1)                 *# think*

*self*.left.take(*self*.number)     *# pickup left chopstick*

            time.sleep(0.1)                 *# (yield makes deadlock more likely)*

*self*.right.take(*self*.number)    *# pickup right chopstick*

            time.sleep(0.1)                 *# eat*

*self*.right.drop(*self*.number)    *# drop right chopstick*

*self*.left.drop(*self*.number)     *# drop left chopstick*

*self*.butler.up()                *# end service by butler*

        sys.stdout.write("p[%s] finished thinking and eating\n" % *self*.number)

* Hàm **main** là hàm chính của chương trình. Biến **n** là số lượng triết gia và đôi đũa. Semaphore **butler** được tạo với giá trị ban đầu là **n-1**. Một danh sách các đôi đũa **c** được tạo. Một danh sách các triết gia **p** được tạo, mỗi triết gia được gán một đôi đũa trái và phải tương ứng và Semaphore **butler**.Sau đó, các triết gia được khởi chạy bằng cách gọi **start()**:

def main():

*# number of philosophers / chop sticks*

    n = 5

*# butler for deadlock avoidance (n-1 available)*

    butler = Semaphore(n-1)

*# list of chopsticks*

    c = [ChopStick(i) for i in range(n)]

*# list of philsophers*

    p = [Philosopher(i, c[i], c[(i+1)%n], butler) for i in range(n)]

    for i in range(n):

        p[i].start()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

* + - * **Qúa trình Wake up và block triết gia:**

A, Quá trình "wake up" (đánh thức) triết gia:

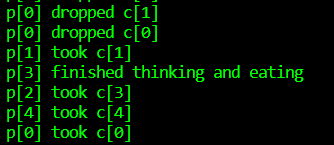
* Triết gia sử dụng phép toán self.butler.down() để yêu cầu butler cho phép mình bắt đầu ăn.
* Trong lớp Semaphore, phương thức down() được gọi, và giá trị của - Semaphore butler được giảm đi 1 (từ 4 xuống còn 3 trong trường hợp này). Nếu giá trị Semaphore là dương, triết gia được phép tiếp tục thực hiện công việc mà không bị block (wake up).
* Nếu giá trị Semaphore là không (0), điều này cho thấy rằng có 3 triết gia khác đang ăn (đã bị block), và tiến trình hiện tại (triết gia mới) sẽ bị block (chặn) và chờ đợi (đánh thức sau khi có Semaphore được giải phóng).

B, Quá trình "block" (chặn) triết gia:

* Triết gia sử dụng phép toán self.left.take(self.number) và self.right.take(self.number) để thực hiện việc lấy đũa bên trái và bên phải.
* Trong lớp ChopStick, phương thức take(self, user) được gọi để lấy đũa. Nếu đũa đã được lấy bởi triết gia khác (biến taken == True), triết gia hiện tại sẽ bị block (chặn) và chờ đợi (đánh thức sau khi có đũa được thả).
  + - * **Kết quả chạy chương trình:**
* Video chạy:



* Giải thích kết quả chạy: giả sử chương trình chạy đến đoạn sau



**+ p[0] dropped c[1]**: Triết gia số 0 đã trả lại đôi đũa có số hiệu 1.

**+ p[0] dropped c[0]**: Triết gia số 0 đã trả lại đôi đũa có số hiệu 0.

**+ p[1] took c[1]**: Triết gia số 1 đã lấy đôi đũa có số hiệu 1.

**+ p[3] finished thinking and eating**: Triết gia số 3 đã hoàn thành suy nghĩ và ăn.

**+ p[2] took c[3]**: Triết gia số 2 đã lấy đôi đũa có số hiệu 3.

**+ p[4] took c[4]**: Triết gia số 4 đã lấy đôi đũa có số hiệu 4.

**+ p[0] took c[0]**: Triết gia số 0 đã lấy đôi đũa có số hiệu 0.

1. **KẾT LUẬN**

* Cả 2 cách đều có thể giải quyết bài toán bữa tối của triết gia, nhưng cần phải chú ý đến các điều kiện để tránh deadlock.
* Sự hiệu quả của hai phương pháp:
  + - 1. Độ phức tạp về mã giả và triển khai: Semaphore cung cấp cấu trúc dữ liệu đồng bộ hóa sẵn có, đi kèm với hai phép toán down và up (mỗi triết gia chỉ cần biết vị trí hai chiếc đũa gần nhất). Điều này làm cho mã giả và triển khai của Semaphore trong bài toán Bữa tối của triết gia đơn giản hơn và dễ hiểu hơn. Về phía khóa (locks), mã giả và triển khai phải dựa trên khóa và các phép toán lock và unlock. Điều này có thể làm cho mã giả và triển khai phức tạp hơn và dễ dẫn đến lỗi nếu không được xử lý cẩn thận.
      2. Tính nhất quán: Semaphore cung cấp một cơ chế đồng bộ hóa mạnh mẽ và linh hoạt hơn, cho phép quản lý quyền truy cập tới tài nguyên chung theo cách tùy chỉnh. Nó đảm bảo rằng chỉ có một số lượng nhất định triết gia được phép lấy đũa cùng một lúc. Với Semaphore, chúng ta có thể dễ dàng kiểm soát và đảm bảo tính nhất quán trong quá trình ăn của các triết gia. Về phía khóa (locks), chúng ta phải cẩn thận quản lý việc đặt và giải phóng khóa để đảm bảo không có triết gia nào lấy cả hai đũa cùng một lúc. Điều này có thể làm tăng khả năng xảy ra tình trạng bế tắc hoặc trạng thái đang đợi trong quá trình ăn của các triết gia.
      3. Hiệu suất và hiệu năng: Semaphore có thể cung cấp hiệu suất tốt hơn trong một số trường hợp. Với Semaphore, chúng ta có thể linh hoạt điều chỉnh số lượng triết gia được phép lấy đũa cùng một lúc, từ đó tối ưu hóa việc sử dụng tài nguyên chung và tránh tình trạng bế tắc. Về phía khóa (locks), nếu không được quản lý cẩn thận, có thể xảy ra tình trạng bế tắc và trạng thái đang đợi, làm giảm hiệu suất của chương trình.
      4. Khả năng mở rộng: Semaphore có tính mô-đun cao, cho phép mở rộng và thay đổi cấu trúc đồng bộ hóa theo nhu cầu – tận dụng được tính chất của biến đếm semaphore, nghĩa là có thể cho phép nhiều hơn một triết gia ăn cùng một lúc nếu số lượng đữa cho phép. Điều này làm cho Semaphore linh hoạt hơn và dễ dàng thích ứng với các yêu cầu mới. Về phía khóa (locks), việc thêm hoặc xóa triết gia có thể phức tạp hơn và đòi hỏi sự xử lý kỹ lưỡng để đảm bảo tính nhất quán và tránh tình trạng bế tắc – vì chỉ cho phép một triết gia ăn tại một thời điểm.
      5. Một số hạn chế: Phương pháp Semaphore có thể gặp khó khăn khi xứ lý ngoại lệ, ví dụ như một triết gia bị ngừng hoạt động khi đang giữa hai chiếc đũa. Phương pháp khóa có thể sử dụng phương pháp timeout hay interrupt để giải quyết các trường hợp này.
* Timeout: Thiết lập một khoảng thời gian
* Interrupt: Một sự kiện hoặc một tín hiệu được gửi đi 🡪 Ngắt !

1. **TÀI LIỆU THAM KHẢO**

[**https://vi.wikipedia.org/wiki/Semaphore\_%28tin\_h%E1%BB%8Dc%29**](https://vi.wikipedia.org/wiki/Semaphore_%28tin_h%E1%BB%8Dc%29)

[**https://viblo.asia/p/tim-hieu-ve-dong-bo-luong-trong-java-voi-bai-toan-bua-toi-cua-cac-triet-gia-OeVKBdMrlkW**](https://viblo.asia/p/tim-hieu-ve-dong-bo-luong-trong-java-voi-bai-toan-bua-toi-cua-cac-triet-gia-OeVKBdMrlkW)

[**https://vimentor.com/vi/lesson/giai-phap-semaphore**](https://vimentor.com/vi/lesson/giai-phap-semaphore)

[**https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/**](https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/)

* Slide môn học Nguyên lý hệ điều hành lớp 141310

**-THE END-**