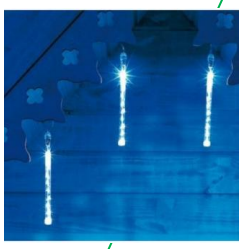
Programmation



La programmation Python

Programmation Asynchrone

Remue-méninges 08

Parallélisme basé sur les "thread"

Bandeau WS2812B de 20 LED

Section de Technicien Supérieur

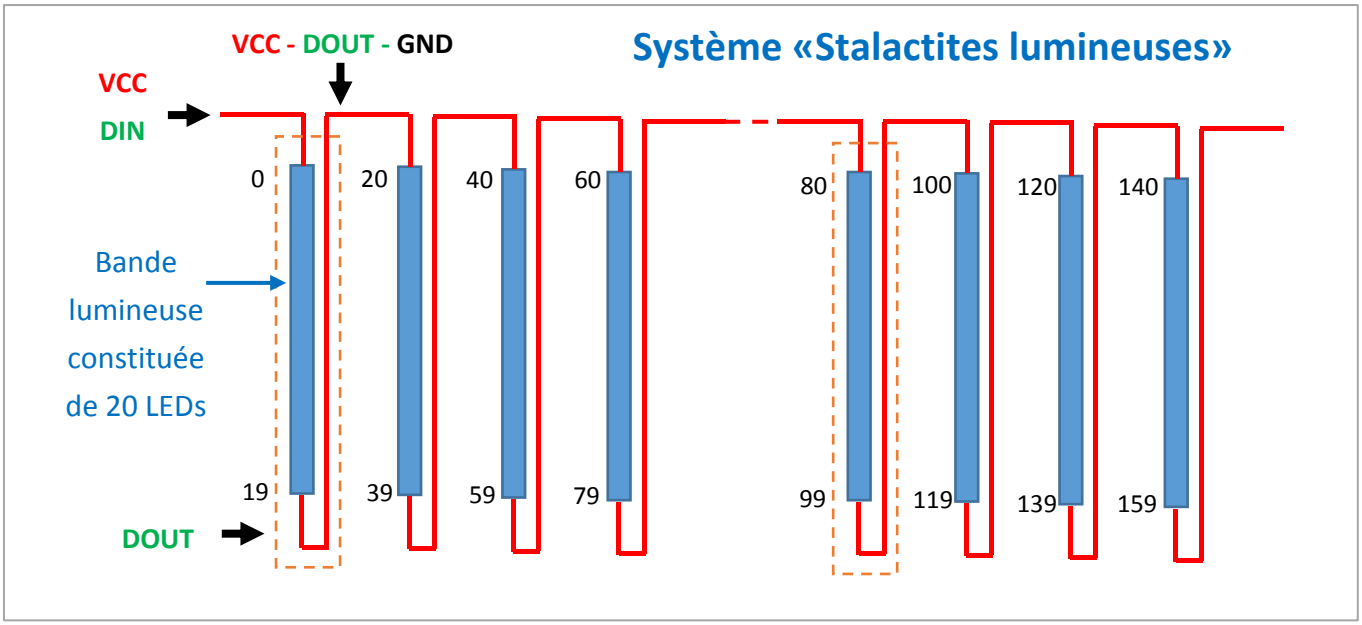
option

Electronique & Communication

A. Contexte

Cet approfondissement concerne la recherche de mise en parallèle d'actions logicielles et s'inscrit dans l'étape de mise au point du système «Stalactites lumineuses».

L'organisation des bandes lumineuses a été définie pour permettre un chaînage des LEDs.



L'utilisation du module "*neopixel*" natif du firmware permet de créer un objet (dans l'exemple : np) où un nombre de LED est défini (dans l'exemple : 20). L'objet permet de définir la situation des 20 LED (de 0 à 19) dans un tableau (np[i]) qui est ensuite à l'aide de la méthode «write» (np.vrite()) transmis sur le bus de données.

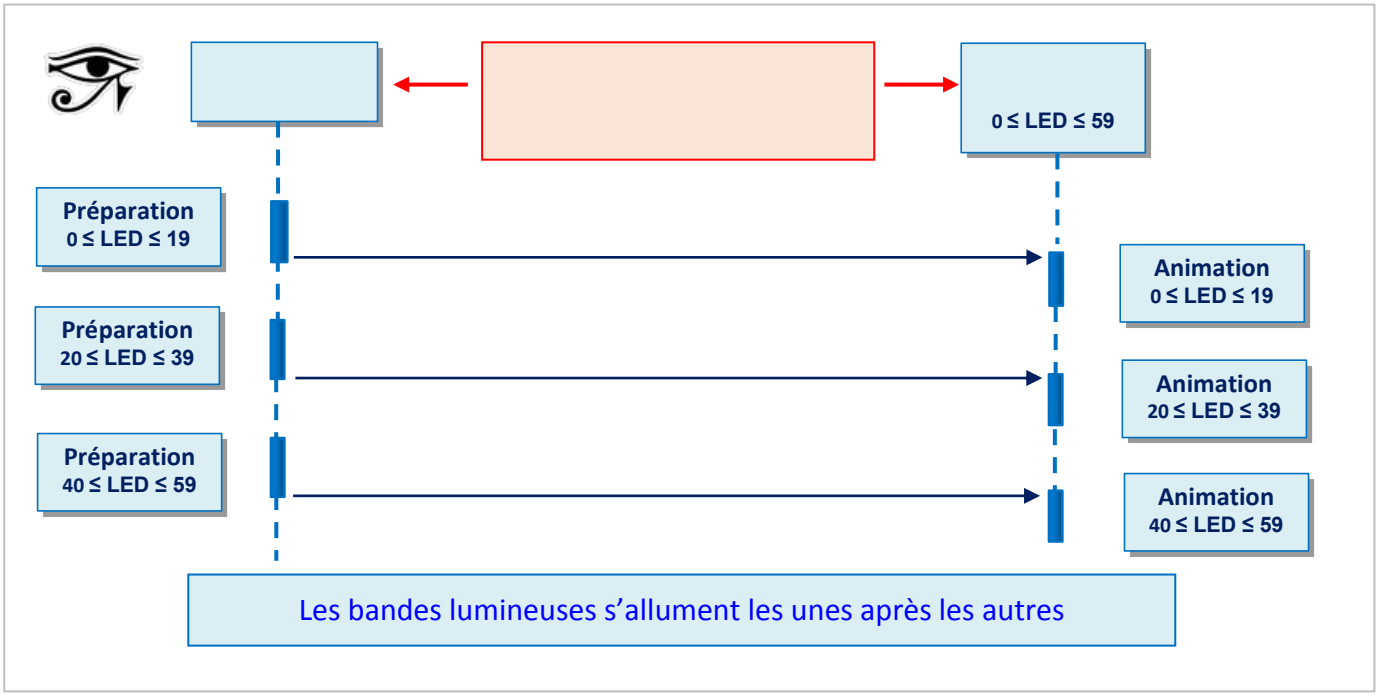
Exemple d'un programme (Effet chenillard) pour UNE bande lumineuse de 20 LED

```
#importation des librairies
import machine, neopixel, time, random
#Nombre de LED sur le ruban
n = 20
#Numéro de la broche
p = 5

#Création d'un objet néopixel sur la broche p
np = neopixel. NeoPixel(machine. Pin(p),n)
#Effet chenillard sur 3 LED
while True :#boucle infinie
    for i in range(20):
        a = random.randint(0,255)
        b = random.randint(0,255)
        c = random.randint(0,255)
        np[i] = (a, b, c)
        np[i-1] = (round(a // 2 ** 1),round(b // 2 ** 1),round(c // 2 ** 1))
        np[i-2] = (round(a // 2 ** 2),round(b // 2 ** 2),round(c // 2 ** 2))
        np[i-3] = (round(a // 2 ** 3),round(b // 2 ** 3),round(c // 2 ** 3))
        np[i-4] = (0, 0, 0)

        np.write()
        time.sleep(0.1)
```

On retient que :

- ✓ Les LED sont chaînées par groupe de **20**,
- ✓ On définit leur couleur en complétant **une ligne de 3 cases** (RGB) d'un **tableau**,
- ✓ La prise en compte **de la totalité** du contenu du tableau est subordonnée à l'exécution de la méthode «write»

Un outil SysML est disponible pour mener une réflexion sur l'architecture algorithmique, il s'agit du **diagramme de séquence**. En effet, le diagramme de séquence permet de décrire comment un cas d'utilisation (ici un processus algorithmique) est réalisé.



Ce processus algorithmique ne permet qu'une animation successive des bandes lumineuses.

Ce n'est pas l'effet recherché puisque l'on souhaite une animation pour chaque bande lumineuse identique (de type stalactite lumineuse) MAIS avec un déclenchement aléatoire sur l'ensemble des bandes lumineuses.

Ce résultat non satisfaisant est lié à l'évolution du programme principal qui réalise les actions les unes après les autres.

Existe-t-il d'autres solutions ?

B. Situation-problème

La programmation dite impérative ou séquentielle réalise l'exécution d'un programme du début jusqu'à la fin ou exprimé différemment de haut en bas. L'interpréteur Python lit ligne à ligne le programme en commençant par la ligne n°1.

Exemple[1] :
programme «*imperatif_1.py*»

Commentaires :
Le processus n°1 est exécuté en premier et complètement, puis le processus n°2.

Remarque :
Pour inverser la présentation des «1» et des «2», il suffit d'inverser les lignes «processus_1()» et «processus()»

Conclusion : La programmation est effectivement séquentielle.

Cette obligation de traiter des événements les uns après les autres peut s'avérer être une difficulté lorsqu'il est nécessaire de gérer des événements issus de situations dont on ne maîtrise pas la temporalité (ils peuvent intervenir à tout moment).

Quel processus algorithmique doit-on programmer pour traiter immédiatement des événements à caractère asynchrone ?

Nous allons explorer l'utilisation des "*thread*".

C. Qu'est ce qu'un "*thread*"

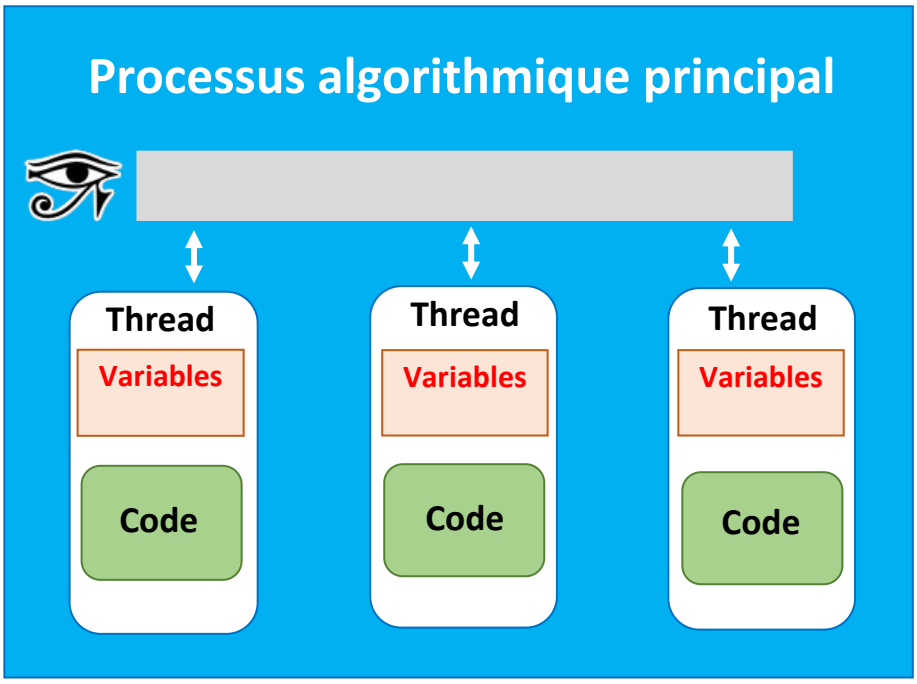
Un "*thread*" que l'on traduira par «fil d'exécution», est une unité d'exécution individuelle et distincte qui s'intègre dans un processus algorithmique. Généralement, plusieurs "*thread*" sont présents dans un processus algorithmique et permettent de donner l'impression d'exécutions parallèles de plusieurs tâches. Cela permet au processus principal de rester réactif aux autres sollicitations :

- ✓ lorsqu'une tâche est réputée longue (par exemple, affichage d'un grand nombre de données), si elle est créée dans un "*thread*", elle s'exécutera à sa vitesse sans bloquer le processus principal,

[1] Exemple inspiré de la vidéo - **Python #28 - programmation asynchrone -** chaîne YouTube de Jason CHAMPAGNE

- ✓ lorsqu'un calcul important peut être scindé en plusieurs calculs indépendants, il sera préférable de créer des "*thread*" pour chaque calcul indépendant afin d'augmenter la rapidité de l'ensemble,
- ✓ lors d'échanges entre un serveur et plusieurs clients, la connexion des clients étant aléatoire et leurs demandes spécifiques, la création d'un "*thread*" pour chaque nouveau client permettra au serveur d'être réactif.

On peut représenter des "*thread*" au sein d'un processus algorithmique de la façon suivante :



Le "*thread*" doit être vu également comme un processus algorithmique mais généralement plus léger en temps d'exécution que le processus principal.
Il possède également un fonctionnement à l'intérieur du processus principal complètement autonome et ne peut partager des ressources que par l'intermédiaire des variables globales.

Les avantages des "*thread*" dans un programme sont donc :

- ✓ La _____ si le système comporte plusieurs processeurs ou si le processeur comportent plusieurs cœurs car dans cette situation, ces "*thread*" seront réellement simultanés,
- ✓ La _____ du système face aux événements extérieurs,
- ✓ L'_____ dans un "*thread*" (par exemple, l'attente d'une information d'entrée au clavier), un autre "*thread*" garde sa capacité d'exécution,
- ✓ La _____ et l'élégance des solutions lorsqu'il faut gérer des parties de programme en concurrence entre eux ou avec des événements par nature asynchrones (par exemple, la connexion d'un client WEB à un serveur WEB).

Parallèlement aux avantages, l'utilisation de plusieurs "*thread*" dans un même processus peut générer des défis :

- ✓ Le partage de ressources lorsqu'il est nécessaire est possible par _____ mais il faut rester vigilant car de fait ces variables interagissent sur l'ensemble des "*thread*",
- ✓ On s'aperçoit que les "*thread*" ont souvent besoin _____, notion de concurrence des "*thread*", ce qui signifie que sans précaution particulière, les résultats peuvent dépendre _____ des "*thread*".

En Python ou en MicroPython, un "*thread*" est un _____. Comme tout objet, il comporte des _____.
Le module nécessaire à l'utilisation de l'objet "*thread*" se nomme "*_thread*".
Il est implémenté nativement [firmware].

```
import _thread
```

D. Mise en oeuvre des "*thread*"

1. Premier exemple : thread_1.py

Gardons l'exemple «*imperatif_1.py*» mais intégrons la notion de programmation asynchrone.
L'objectif est de garder les deux processus permettant chacun d'afficher un message différent mais en initiant un fonctionnement que l'on souhaite simultané.

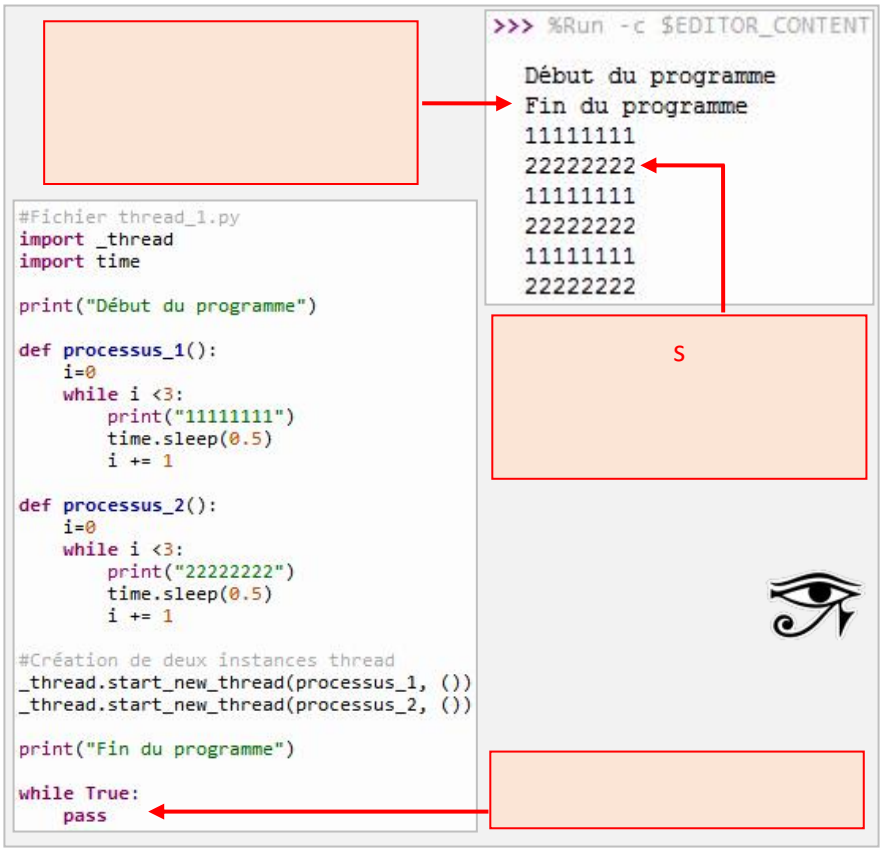
Pour cela, chaque fonction va être liée à un "*thread*".

Commentaires :

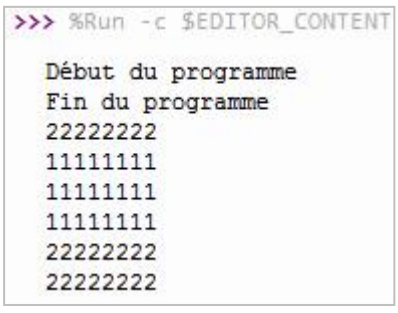
_____

_____

Remarque :
Que se passe-t-il si une durée d'attente est plus importante dans un processus ?



Dans la fonction «processus_2», écrivons _____. On obtient :



Conclusion :
Le "*thread*" _____
_____. Le blocage réalisé par la temporisation de 4s n'est pas

_____ pour l'exécution du processus_1.

Attention : les "*thread*" sont toujours en fonctionnement dans la boucle «while True»

2. Deuxième exemple : thread_2.py

L'objectif de ce deuxième exemple est de préparer les conditions permettant d'arrêter le programme principal.

Pour cela, deux variables globales sont déclarées et créées :

- ✓ **nb_thread** qui permet de définir le nombre de "thread" à activer,
- ✓ **nb_thread_fini** qui quantifie le nombre de "thread" à partir duquel le programme sera stoppé

Remarque importante :

Pour utiliser une variable globale dans une fonction, _____ _____.

Remarque :

Chaque "thread", à sa création, possède un identifiant. Cet identifiant peut être communiqué en utilisant la méthode «get_ident()»

```
#Fichier thread_2.py
import _thread
import time

print("début du programme")

def processus_1():
    i=0
    while i <3:
        print(str(_thread.get_ident()) + " : " + "11111111")
        time.sleep(0.5)
        i += 1
    global nb_thread_fini
    nb_thread_fini += 1

def processus_2():
    i=0
    while i <3:
        print(str(_thread.get_ident()) + " : " + "22222222")
        time.sleep(0.5)
        i += 1
    global nb_thread_fini
    nb_thread_fini += 1

#Nombre de thread prévu
global nb_thread
nb_thread = 2
#Nombre de thread après réalisation
global nb_thread_fini
nb_thread_fini = 0

#Création de deux instances thread
_thread.start_new_thread(processus_1, ())
_thread.start_new_thread(processus_2, ())

#Boucle d'attente de fin d'exécution des thread
while nb_thread_fini < nb_thread:
    pass
print("Fin du programme")
```

```
>>> %Run -c $EDITOR_CONTENT

  début du programme
  1073538828 : 11111111
  1073543948 : 22222222
  1073538828 : 11111111
  1073543948 : 22222222
  1073538828 : 11111111
  1073543948 : 22222222
  Fin du programme

>>>
```

3. Troisième exemple : thread_3.py

L'objectif de ce troisième exemple est de créer un plus grand nombre de processus pouvant être commandé par des "thread" et d'analyser la succession des résultats.

```python
#Fichier thread_3.py
import _thread
import time

print("début du programme")

#Enregistrement des messages des processus dans une variable globale
message = ""

def processus(id, nbr):
    i=0
    while i <3:
        global message
        message += str(_thread.get_ident()) + " : " + str(nbr) + "\r\n"
        time.sleep(0.5)
        i += 1
    global nb_thread_fini
    nb_thread_fini += 1

#Nombre de thread prévu
nb_thread = 4
#Nombre de thread après réalisation
nb_thread_fini = 0

#Création de quatre instances thread
for th in range(1, nb_thread + 1):
    _thread.start_new_thread(processus, (th, 11111111*th))

#Boucle d'attente de fin d'exécution des thread
while nb_thread_fini < nb_thread:
    pass
print(message)
print("Fin du programme")
```

```
>>> %Run -c $EDITOR_CONTENT

  début du programme
  1073543948 : 22222222
  1073554204 : 44444444
  1073538828 : 11111111
  1073549068 : 33333333
  1073538828 : 11111111
  1073543948 : 22222222
  1073554204 : 44444444
  1073549068 : 33333333
  1073538828 : 11111111
  1073543948 : 22222222
  1073554204 : 44444444
  1073549068 : 33333333

  Fin du programme

>>>
```

Conclusion :

On observe bien une progression aléatoire comme l'illustre un deuxième essai ci contre :

```
>>> %Run -c $EDITOR_CONTENT

  début du programme
  1073543948 : 22222222
  1073538828 : 11111111
  1073549068 : 33333333
  1073554204 : 44444444
  1073538828 : 11111111
  1073543948 : 22222222
  1073549068 : 33333333
  1073554204 : 44444444
  1073538828 : 11111111
  1073549068 : 33333333
  1073543948 : 22222222
  1073554204 : 44444444

  Fin du programme

>>>
```

4. Quatrième exemple : thread_4.py

L'objectif de ce quatrième exemple est de mettre en oeuvre le concept de verrouillage.
Il est utile lorsque l'on souhaite synchroniser un "thread" par rapport au programme principal ou par rapport aux autres "thread".

Cela s'effectue en plusieurs étapes.

Dans le programme principal :

✓ _____

```
verrouillage = _thread.allocate_lock()
```

✓ _____

```
verrouillage.acquire()
```

Dans le "thread" que l'on souhaite bloquer :

✓ _____

```
verrouillage.acquire()
```

De nouveau dans le programme principal ou dans un autre "thread" que celui qui est bloqué :

✓ _____

```
verrouillage.release()
```

Le programme suivant consiste à bloquer le "thread" n°1 jusqu'à l'exécution des "thread" n°2, 3 et 4.
Après l'exécution de ces "thread", le "thread" n°1 est libéré.

```python
#Fichier thread_4.py
import _thread
import time

print("début du programme")

#Enregistrement des messages des processus dans une variab
message = ""

def processus(id, nbr):
    if id == 1:
        verrouillage.acquire()
    i=0
    while i <3:
        global message
        message += str(_thread.get_ident()) + " : " + str(n
        time.sleep(0.5)
        i += 1
    global nb_thread_fini
    nb_thread_fini += 1

#Nombre de thread prévu
nb_thread = 4
#Nombre de thread après réalisation
nb_thread_fini = 0

#Création d'un objet "verrouillage"
verrouillage = _thread.allocate_lock()
verrouillage.acquire()

#Création de quatre instances thread
for th in range(1, nb_thread + 1):
    _thread.start_new_thread(processus, (th, 11111111*th))

#Boucle d'attente de fin d'exécution des thread
while nb_thread_fini + 1 < nb_thread:
    pass
verrouillage.release()
while nb_thread_fini < nb_thread:
    pass
print(message)
print("Fin du programme")
```

```
>>> %Run -c $EDITOR_CONTENT

  début du programme
  1073543948 : 22222222
  1073554204 : 44444444
  1073538828 : 11111111
  1073549068 : 33333333
  1073538828 : 11111111
  1073543948 : 22222222
  1073554204 : 44444444
  1073549068 : 33333333
  1073538828 : 11111111
  1073543948 : 22222222
  1073554204 : 44444444
  1073549068 : 33333333

  Fin du programme

>>>
```

```
>>> %Run -c $EDITOR_CONTENT

  début du programme
  1073543948 : 22222222
  1073549068 : 33333333
  1073554204 : 44444444
  1073543948 : 22222222
  1073554204 : 44444444
  1073549068 : 33333333
  1073554204 : 44444444
  1073543948 : 22222222
  1073549068 : 33333333
  1073538828 : 11111111
  1073538828 : 11111111
  1073538828 : 11111111

  Fin du programme

>>>
```

E. Mise en application des "*thread*" dans le cadre du projet «Stalactites lumineuses»

1. Recherche d'une structure algorithmique

- ✓ Envisageons d'utiliser 10 "thread" ainsi chaque "thread" gère 2 bandes lumineuses non contiguës dans le tableau,
- ✓ Chaque "thread" complète, à son rythme, la partie du tableau concernée par les 2 bandes lumineuses et agit sur la méthode «write» pour provoquer l'affichage.