

# Multiprocessing - S7

---

Dans le cadre du module "Système d'exploitation et programmation concurrente", nous avons eu à réaliser différents exercices portant sur le principe de la parallélisation de processus (multiprocessing). Pour ce faire nous avons dû réaliser différents exercices chacun rapportant un certain nombre de point. Comme nous sommes trois dans notre groupe nous devons faire pour 30 points minimum d'exercice.

Suite aux recommandations des professeurs référents de la matière, les exercices portants sur le tri (tri rapide, tri merge ...) n'ont pas été réalisés car jugés comme s'éloignant trop de la logique du multiprocessing abordée en cours.

Fait en trinôme : 4-IRC

- BALAGUER Eileen
- BATTU Maxime
- LALOI Batiste

## Exercices réalisés

---

### Courses Hippique (3pts)

Récupération du code donné, étude et compréhension de ce dernier. Plusieurs tests effectués pour comprendre le multiprocessing et son intérêt dans cet exercice. Après quelques manipulations nous avons pris le code en main et rajouter différentes modifications au code donné.

Travail réalisé

- Mise en place d'un processus arbitre qui affiche en permanence le cheval qui est en tête ainsi que celui qui est dernier
  - stack trace :

```
Leader: [Lettre du leader] Dernier : [Lettre du dernier]
```

- Faire un pari sur un cheval gagnat
  - Entrer une lettre entre A et T :
- Modification du dessin de base pour le remplacer par un bateau
  - +|\_\_A\_\_|

### Faites des calculs (calculateurs & demandeurs) (3-5pts)

Version 1 demandeur, n calculateurs

Le programme principal crée un certain nombre de processus "calculateurs" (par défaut, 2) qui attendent des expressions de calcul à résoudre dans une file d'attente (Queue). Un autre processus, appelé "demandeur", génère des expressions de calcul aléatoires et les met dans la file d'attente (Queue) pour être résolues par les

processus "calculateurs". Les processus "calculateurs" résolvent les expressions de calcul et mettent le résultat dans une autre file d'attente (Queue). Le processus "demandeur" récupère les résultats des processus "calculateurs" et les affiche à l'écran. Le programme se termine lorsque tous les calculs ont été effectués.

- exemple de stack trace :

- ```
Combien de calculs voulez-vous lancer ? 2 par défaut

Combien de processus calculateurs ? 2 par défaut

Le fils a reçu 3+4
Dans fils, le résultat = 7
Le fils a envoyé 7
3+4 = 7
```

## Gestionnaire des Billes (5pts)

Cet exercice était assez guidé. On comprend qu'on a N joueurs (processus) qui ont chacun besoin d'un nombre k de ressources. Donc je commence mon code par demandé le nombre de joueur N et pour chaque joueur le nombre k de ressources nécessaires pour lui.

Ensuite je demande le nombre d'itérations pour pouvoir répéter m fois la séquence.

Je mets en place une variable protégée pour permettre aux joueurs d'accéder aux billes chacun leur tour, évitant ainsi de se retrouver avec -8 billes.

Je crée un processus Contrôleur qui, toutes les secondes, va vérifier que mon nombre de billes est supérieur à 0 et inférieur au nombre maximum de billes disponible.

Une fois fait, j'ai créé mes processus joueur qui répète la séquence m fois de "prendre ; utiliser ; rendre".

Mon "prendre" est représenté par une fonction qui utilise la variable protégée pour prélever le nombre de billes voulues dans le stock quand personne n'agit avec. Mon "utiliser" est représenté par un `time.sleep(2)` qui va donner 2 secondes de temps d'attente. Mon "rendre" est représenté par une fonction qui utilise la variable protégée pour rendre le nombre de billes voulues dans le stock quand personne n'agit avec.

- stack trace :

```
Nombre de processus : 3
Nombre d'itérations : 2
Nombre max de billes : 8
Ressources requises : 4
Ressources requises : 3
Ressources requises : 5
Process n° 1 demande. Ressources dispo : 8
Process n° 1 a reçu. Ressources dispo : 5
Process n° 1 utilise ses ressources
Process n° 2 demande. Ressources dispo : 8
Process n° 0 demande. Ressources dispo : 8
```

```
Process n° 0 a reçu. Ressources dispo : 4
Process n° 0 utilise ses ressources
Process n° 2 a reçu. Ressources dispo : 3
Process n° 2 utilise ses ressources
Process n° 1 rends. Ressources dispo : 5
Process n° 1 a rendu. Ressources dispo : 8
Process n° 1 demande. Ressources dispo : 8
Process n° 1 a reçu. Ressources dispo : 5
Process n° 1 utilise ses ressources
Process n° 0 rends. Ressources dispo : 4
Process n° 0 a rendu. Ressources dispo : 8
Process n° 0 demande. Ressources dispo : 8
Process n° 0 a reçu. Ressources dispo : 4
Process n° 0 utilise ses ressources
Process n° 2 rends. Ressources dispo : 3
Process n° 2 a rendu. Ressources dispo : 8
Process n° 2 demande. Ressources dispo : 8
Process n° 2 a reçu. Ressources dispo : 3
Process n° 2 utilise ses ressources
Process n° 1 rends. Ressources dispo : 5
Process n° 1 a rendu. Ressources dispo : 8
Process n° 0 rends. Ressources dispo : 4
Process n° 0 a rendu. Ressources dispo : 8
Process n° 2 rends. Ressources dispo : 3
Process n° 2 a rendu. Ressources dispo : 8
```

## Estimation de PI

Le but ici était de réussir à estimer la valeur de PI à partir de différentes techniques mathématiques. Nous devons donner un grand nombre d'itérations (100 000 000 dans notre cas) pour se rapprocher au plus possible de la valeur de PI.

Un code nous était donné, cependant il était "mono-processus" nous devons alors le modifier pour mettre en place du multiprocessing à la place

### Version Hit-Miss Monte Carlo (3pts)

Cette technique mathématique sert à déterminer la surface d'un quart du cercle trigonométrique et d'ensuite multiplier le résultat obtenu par le nombre de quart contenu dans un cercle, grâce à quoi nous pouvons obtenir une approximation de PI.

- contient un main qui appelle 2 différentes méthodes
  - multiprocessing(nbliterations)
    - Découper le nombre d'itération par le nombre de processus
    - Appeler les différents processus avec un nombre d'itérations calculé précédemment
    - Récupérer le résultat mis dans une Queue
    - Estimer la valeur de pi
  - multiprocessing(nbliterations)
    - Le calcul mathématique est le même nous utilisons simplement 1 seul et unique processus qui devrait effectué le calcul hit-miss pour les N estimations contrairement au code

### multiprocessus

- stack trace :

```
Début du multiprocessing
Temps de traitement XX.XX secondes pour X iterations en multiprocess
Valeur estimée Pi par la méthode Hit-Miss avec 4 processus : X.XXXXXXX
Fin du multiprocessing

Début du monoprocessus
Temps de traitement XX.XX secondes pour X iterations en monoprocessus
Valeur estimée Pi par la méthode Hit-Miss en mono-processus : X.XXXXXXX
Fin du monoprocessus
```

### Version Arc-tangente (3pts)

Même principe que la première estimation de pi avec une technique mathématique différente.

- stack trace :

```
Valeur estimée Pi par la méthode arc-tangente en multiprocess :
X.XXXXXXX
Temps de traitement XX.XX secondes pour X iterations en monoprocessus
```

### Version par l'espérance (3pts)

Même principe que la première estimation de pi avec une technique mathématique différente.

- stack trace :

```
Valeur estimée Pi par la méthode arc-tangente avec X processus :
X.XXXXXX...
Temps de traitement XX.XX secondes pour X iterations en multiprocess
```

## Un système multi-tâches de simulation d'un restaurant (5pts)

Pour cet exercice je l'ai découpé en trois parties : client ; serveur ; majorHomme Je commence par demander dans cet exercice le nombre de serveurs qui vont pouvoir prendre des commandes et le nombre de commande à traiter.

La partie cliente s'occupe de créer les commandes à traiter. Pour cela, il va rajouter une commande (concaténation entre un chiffre et une lettre) toutes les 1 à 4 secondes.

Les serveurs vont prendre chacun leur tour une commande dans la liste de celles qui sont en attente, ils mettront entre 3 et 5 secondes pour la traiter et une fois fait ils remplissent une variable qui indiquera la dernière commande servie.

Le majorHomme quant à lui va venir afficher tout ça. Il va afficher 1 ligne par serveur qui va indiquer la commande qu'il est en train de traiter. Ensuite il indiquera les commandes qui sont dans le tableau d'attente, le nombre qu'elles sont, et la dernière commande servie.

Tout du long de cet exercice j'ai utilisé un Locker pour éviter que deux processus écrivent en même temps sur mes tableaux et donc limiter les erreurs dans le traitement des commandes

- Exemples de stack trace:

```
Nombre de serveurs : 3
Max de commande possible : 8

Le serveur 1 traite la commande 7I
Le serveur 2 ne traite pas de commande pour le moment
Le serveur 3 ne traite pas de commande pour le moment

Les commandes clients en attente : ['8I', '46W']
Nombres de commandes attente : 2
Commande 37N est servie au client
```

```
Nombre de serveurs : 4
Max de commande possible : 10

Le serveur 1 traite la commande 34F
Le serveur 2 ne traite pas de commande pour le moment
Le serveur 3 ne traite pas de commande pour le moment
Le serveur 4 ne traite pas de commande pour le moment

Les commandes clients en attente : ['14T']
Nombres de commandes attente : 1
Commande 25R est servie au client
```

## Température et pression

Mise en place un système temps réel embarqué pour maintenir la température et la pression d'un processus chimique dans des limites spécifiées. Le système comprend plusieurs entités concurrentes, un contrôleur pour coordonner l'ensemble, et une zone mémoire partagée protégée par un verrou ou un sémaphore.

- Les entités :
  - Contrôleur
    - Contrôle les valeurs récupérées **température** et **pression** afin de déterminer si les différents acteurs du système doivent s'arrêter ou continuer de fonctionner
    - Acteurs
      - **Pompe** mesure la pression dans le système
      - **Chauffage** mesure la température dans le système

Pour réaliser ça nous avons créé plusieurs variables partagées permettant d'avoir une information partagée entre plusieurs processus et un locker pour protéger la zone de mémoire.

- Stack trace :

- Température : 11.34 °C  
 Pression : 1.06 bar  
  
 Chauffage : ON  
 Pompe : ON

**Ctrl + c | Ctrl + Z pour arrêter le programme**

## Fractales (3 pts)

Récupération du code mono-process donné, et adaptation en multi-process

Travail réalisé

- Transformation du code en programme multi-process grâce à la fonction `multiProcess()` qui utilise un pack de processus qui vont gérer `NB_PROCESS` de lignes à la fois, puis effectuer un roulement pour faire le rendu des prochaines lignes disponibles. Ajout d'une barre de chargement pour afficher l'avancé du rendu :

```
def multiProcess():
    # Créer un pool de processus
    for pack in range(size//NB_PROCESS):
        # On les lance chacun sur la prochaine ligne à render
        # Prochaine ligne à traiter = pack * NB_PROCESS + line
        for line in range(NB_PROCESS):
            p = mp.Process(target=render_line, args=(pack*NB_PROCESS+line, image))
            p.start()

        # On attends bien la fin de tous les processus
        for line in range(NB_PROCESS):
            p.join()

        # Afficher la progression du rendu sous la forme d'une barre de chargement
        percentCompletion = round(pack * 100 / (size//NB_PROCESS))
        print("[ " + "=" * (percentCompletion//2) + " " * (50 -
percentCompletion//2) + "]" + str(percentCompletion) + "%", end="\r")
```

- Chacun de ces processus pointe sur une fonction `render_line()` qui traite chaque cellule d'une ligne, en appliquant la fonction :

```
# Fonction qui traite une ligne
def render_line(line, image):
    for x in range(size):
        calcul(x, line, image, 3*(line * size + x))
```

- Récupération du nombre de process voulu par l'utilisateur, avec affichage du nombre de coeur disponible sur sa machine :

```
print(f"Image de taille {size}x{size}")
NB_PROCESS = int(input(f"Entrer un nombre de processus (Entre 1 et
{mp.cpu_count()}): "))
print(f"Rendu de l'image en cours avec {NB_PROCESS} processus...")
```

- Calcul du temps de rendu de l'image :

```
import time

# Démarrage du timer
start_time = time.time()

# Lancement du programme de rendu de l'image
multiProcess()

# Fin du timer
end_time = time.time()

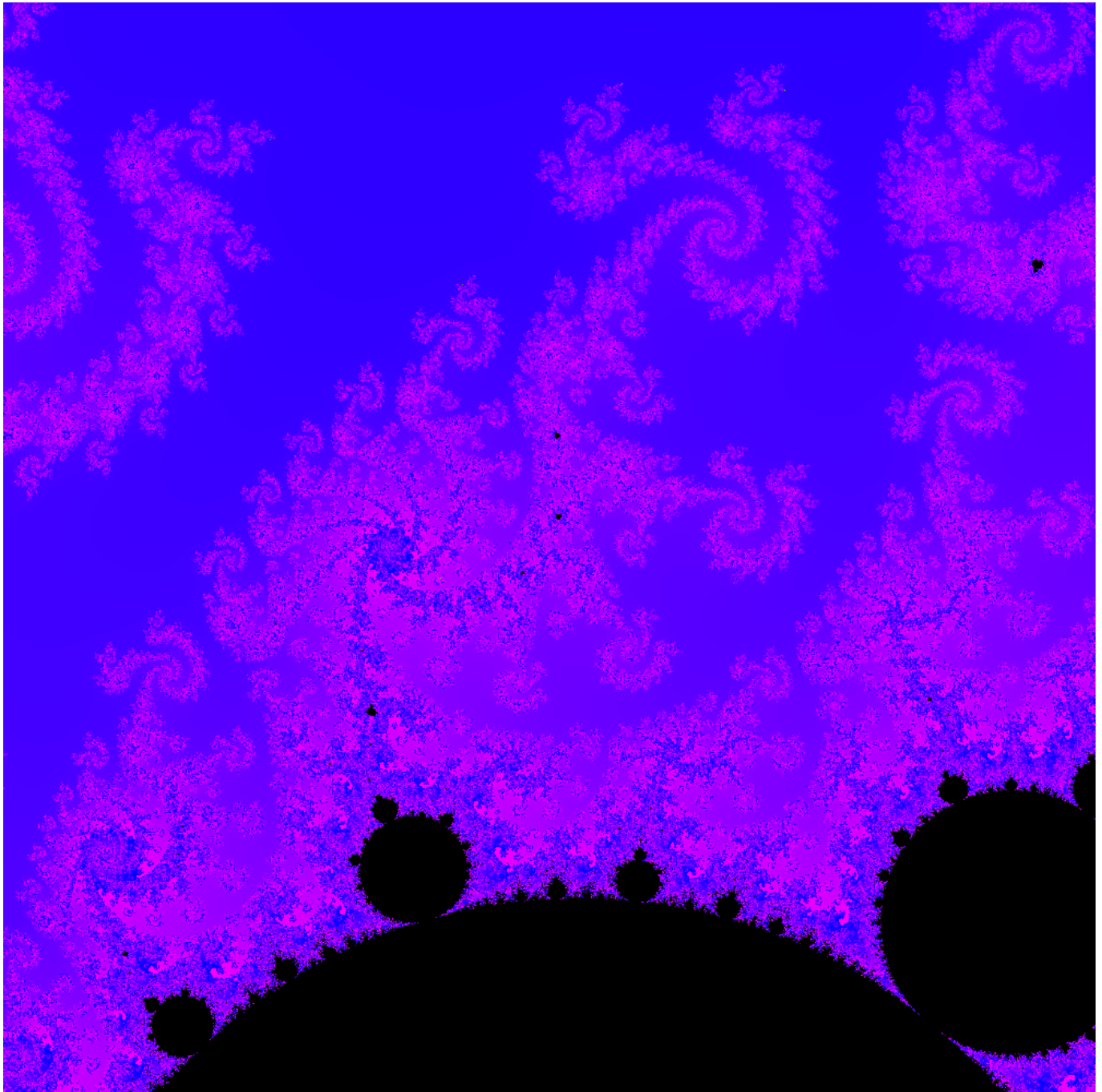
# Affichage du temps de rendu, avec deux chiffres après la virgule
print(f"\nTemps d'exécution: {end_time - start_time:.2f} secondes")
```

## Trace

Voici l'affichage en console lors de l'exécution du programme :

```
$> python3 fractal.py
Image de taille 1000x1000
Entrer un nombre de processus (Entre 1 et 16): 10
Rendu de l'image en cours avec 10 processus...
[===== ] 55%
```

Résultat du rendu de l'image fractale en 1000x1000 (2K) :



## Game Of Life (5 pts)

Création d'un clone du Jeu de la Vie créé par Horton Conway, en utilisant du multiprocessing

Travail réalisé

- Création d'un pack de **NB\_PROCESS** processus qui effectuent un roulement de la même manière que le programme des **Fractales** :

```
NB_PROCESS = 5

# Pack de processus
for pack in range(TAILLE//NB_PROCESS):
    for line in range(NB_PROCESS) :
        p = mp.Process(target=render_line, args=(pack*NB_PROCESS + line,
```



```
current_grid, next_grid))
    p.start()

    for _ in range(NB_PROCESS):
        p.join()
```

- Création de deux grilles partagées, représentant la génération actuelle et la prochaine. Création de deux tableaux à deux dimensions grâce à la bibliothèque **Numpy**:

```
TAILLE = 30

# Création de deux tableaux à deux dimensions partagés
shared_current_grid = mp.Array(ctypes.c_int, TAILLE * TAILLE)
shared_next_grid = mp.Array(ctypes.c_int, TAILLE * TAILLE)

# Création d'un tableau numpy à partir des données partagées
current_grid = np.frombuffer(shared_current_grid.get_obj(), ctypes.c_int)
next_grid = np.frombuffer(shared_next_grid.get_obj(), ctypes.c_int)

# Raformer le tableau en un tableau à deux dimensions
current_grid = current_grid.reshape((TAILLE, TAILLE))
next_grid = next_grid.reshape((TAILLE, TAILLE))
```

- Définition d'une fonction d'affichage de la grille :

```
def display_grid(grid):
    # Affichage d'une ligne en haut pour encadrer la grille
    print("_" * (2 * TAILLE + 1))

    for line in grid:
        # Affichage d'une ligne à gauche pour encadrer la grille
        print("|", end="")

        # Boucle d'affichage
        for cell in line:
            if cell == 0:
                print(" ", end="")
            else:
                print("■", end="")

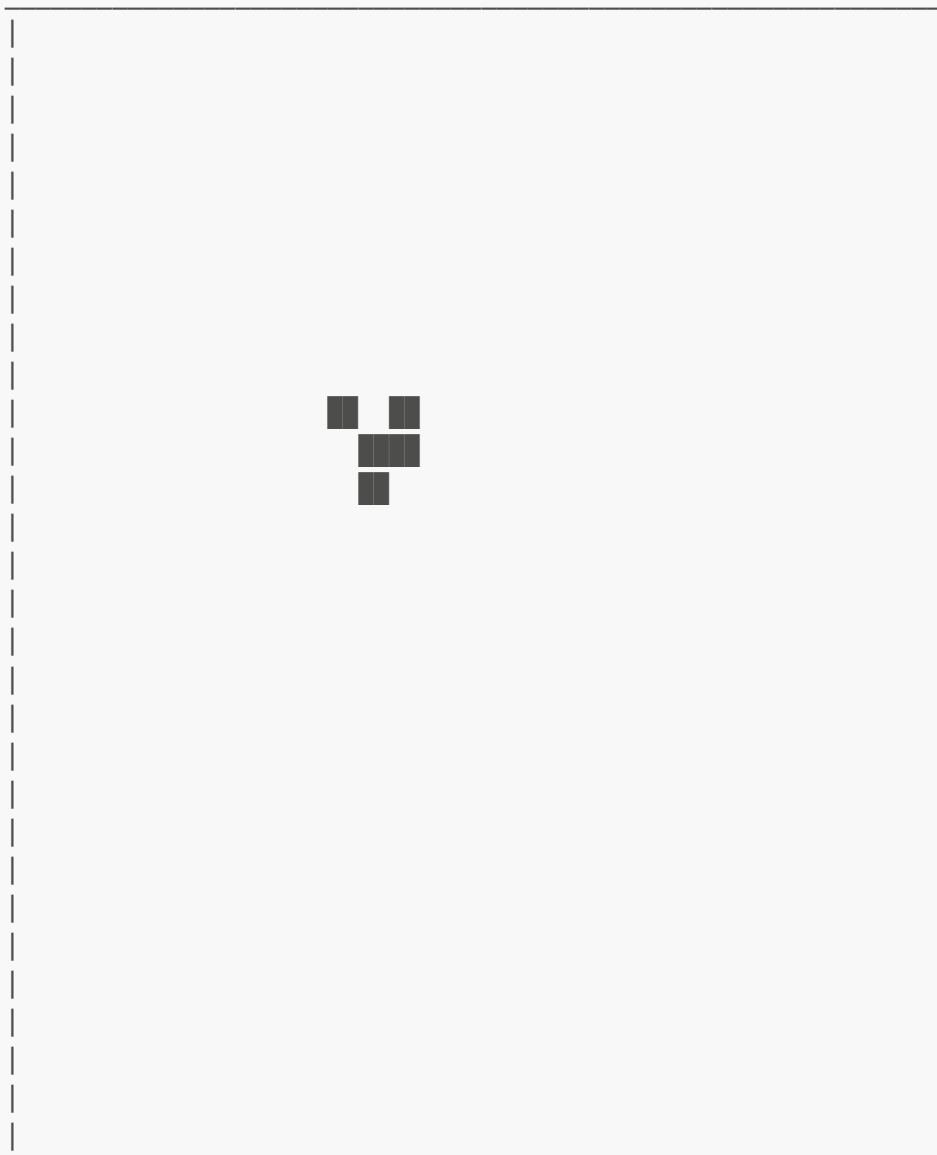
        # Affichage d'une ligne à droite pour encadrer la grille
        print("|")

    # Affichage d'une ligne en bas pour encadrer la grille
    print("-" * (2 * TAILLE + 1))
```

- Fonction d'ajout d'un "glider" (un vaisseau qui se propage), pour tester le bon fonctionnement des règles du jeu :

```
def addGlider(x, y, grid):  
    glider = np.array([[0, 0, 1], [1, 0, 1], [0, 1, 1]])  
    grid[x:x+3, y:y+3] = glider  
  
# Ajout d'une glider dans la grille de départ à la case [1:1]  
addGlider(1, 1, current_grid)
```

Affichage de la grille avec le vaisseau après 35 générations :



Génération numéro : 35

- Rendu final : Remplissage aléatoire de la grille de départ :

```
import random  
  
# On remplit de manière aléatoire la grille, avec 20% de chance d'avoir une
```

```

cellule vivante
for i in range(TAILLE):
    for j in range(len(current_grid[i])):
        if random.randint(0, 4) == 0:
            current_grid[i][j] = 1

```

- Fonctions de gestions des règles et création de la prochaine génération de cellules :

```

# Fonction qui renvoie le nombre de voisins vivants d'une cellule donnée
def living_neighbours_count(x, y, grid):
    count = 0
    for i in range(-1, 2):
        for j in range(-1, 2):
            if i == 0 and j == 0:
                continue
            elif x + i < 0 or x + i >= TAILLE or y + j < 0 or y + j >= TAILLE:
                continue
            elif grid[x + i][y + j] == 1:
                count += 1
    return count

# Fonction qui traite une cellule
def render_cell(x, y, current_grid, next_grid):
    living_neighbours = living_neighbours_count(x, y, current_grid)
    if current_grid[x][y] == 1:
        if living_neighbours == 2 or living_neighbours == 3:
            next_grid[x][y] = 1
        else :
            next_grid[x][y] = 0
    else:
        if living_neighbours == 3:
            next_grid[x][y] = 1

# Fonction qui traite une ligne à la fois, exécutée par les processus
def render_line(line, current_grid, next_grid):
    for cell in range(TAILLE):
        render_cell(line, cell, current_grid, next_grid)

```

- Boucle de jeu :

```

while True:

    # Effacer l'écran
    move_to(0, 0)

    # Pack de processus
    for pack in range(TAILLE//NB_PROCESS):

```

```
        for line in range(NB_PROCESS) :
            p = mp.Process(target=render_line, args=(pack*NB_PROCESS + line,
current_grid, next_grid))
            p.start()

        for _ in range(NB_PROCESS):
            p.join()

display_grid(next_grid)

# Actualisation de la grille actuelle avec la nouvelle génération
for i in range(TAILLE):
    for j in range(len(next_grid[i])):
        current_grid[i][j] = next_grid[i][j]

cpt += 1
print(f"Génération numéro : {cpt}")
```

Trace :

Comme le rendu dans le console est dynamique, voici une vidéo de l'affichage dans la console, avec à gauche le jeu de la vie qui tourne dans une grille de 30x30, et à droite un affichage de HTOP, dans lequel on peut voir le charge réparties sur les différents coeurs du processeur de l'ordinateur : [Démonstration Game Of Life](#)