Courses Hippique (3pts)

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

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 gagnant
  - `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 recu 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 en demandant le nombre de joueurs N, et pour chaque joueur, le nombre k de ressources nécessaires pour lui.

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

Je mets en place une variable protégée pour permettre aux joueurs d'accéder aux ressources chacun leur tour, évitant ainsi de se retrouver avec un nombre négatif de ressources.

Je crée un processus contrôleur qui, toutes les secondes, vérifie que le nombre de ressources est supérieur à 0 et inférieur au nombre maximum de ressources disponibles.

Une fois fait, j'ai créé mes processus joueur qui répètent 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 ressources voulu dans le stock quand personne n'agit avec. Mon "utiliser" est représenté par un time.sleep(2) qui va donner 2 secondes d'attente. Mon "rendre" est représenté par une fonction qui utilise la variable protégée pour rendre le nombre de ressources voulu 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 est d'estimer la valeur de PI à l'aide de différentes techniques mathématiques. Nous devons donner un grand nombre d'itérations (100 000 000 dans notre cas) pour se rapprocher au plus près de la valeur de PI.

Un code nous a été donné, mais il était "mono-processus", nous devons donc 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 de cercle trigonométrique, puis à multiplier le résultat obtenu par le nombre de quarts contenus dans un cercle complet pour obtenir une approximation de PI.

- contient un main qui appelle 2 différentes méthodes
  - multiprocess(nbIterations)
    - 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
  - monoprocess(nbIterations)
    - 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 mu monoprocessus
```

Version Arc-tangante (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, j'ai découpé le code en trois parties : client, serveur et majorHomme.

Je commence par demander le nombre de serveurs et le nombre de commandes à traiter.

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

Les serveurs prennent chacun leur tour une commande dans la liste des commandes en attente, et mettent entre 3 et 5 secondes pour la traiter. Une fois la commande traitée, ils remplissent une variable indiquant la dernière commande servie.

Le majorHomme affiche les informations. Il affiche une ligne par serveur, indiquant la commande en cours de traitement. Ensuite, il affiche les commandes en attente, leur nombre, et la dernière commande servie.

Tout au long de l'exercice, j'ai utilisé un verrou (locker) pour éviter que deux processus écrivent simultanément dans mes tableaux, et ainsi éviter 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

Pour maintenir la température et la pression d'un processus chimique dans des limites spécifiées, j'ai mis en place un système temps réel embarqué. Ce 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 :
  - Controller
    - Controle les valeurs récupérées `température` et `pression` afin de déterminer si les différents acteurs du système doivent s'arreter ou continuer de fonctionner
    - Acteurs
      - `Pompe` mesure la pression dans le système
      - `Chauffage` mesure la température dans le système

Pour réaliser cela, nous avons créé plusieurs variables partagées pour permettre l'accès à des informations par plusieurs processus, ainsi qu'un verrou 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-processus grâce à la fonction multiProcess() qui utilise un groupe de processus pour gérer NB_PROCESS lignes simultanément, puis effectue un roulement pour traiter les lignes suivantes. J'ai ajouté une barre de progression pour afficher l'avancement 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'éxécution 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%
```

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, représentatnt 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)
```

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

```
import random

# On remplis 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, executée 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}")
```