TP 1 : Utilisation du module `threading`

Exercice 1.1

1. Définissez une classe `Counter` dont chaque instance est un moniteur pour un compteur distinct. Outre la méthode standard `__init__` qui doit permettre la création d'une instance et l'initialisation de la valeur du compteur, la classe doit fournir les méthodes suivantes :
   - `inc` qui incrémente le compteur,
   - `dec` qui décrémente le compteur.
   - `value` qui retourne la valeur actuelle du compteur.

   L'implémentation de ces méthodes doit garantir un accès exclusif au compteur.
2. Écrivez une fonction de profil `worker(counter, up, num_op)` où `counter` est une référence à un objet de la classe `Counter`, `up` est un booléen et `num_op` spécifie le nombre fois où le compteur doit être incrémenté (cas de `up` vrai) ou décrémenté (cas de `up` faux).
3. Utilisez le module `random` pour créer des threads excécutant la fonction `worker` avec des valeurs choises au hasard, sauf un compteur unique partagé par tous les threads référencé par `counter`.
   - Le nombre des threads est choisi par `random.randint(1,16)`.
   - Chaque thread exécute la fonction `worker` avec le paramètre `up` choisi par
     `random.choice([True, False])`
     et le paramètre `num_op` choisi par `random.randint(1,16)`. En plus de `target=worker`, les arguments à passer à `worker` sont donnés à la création du thread via `args=(counter, ...)`.
4. Pour vous assurer du bon fonctionnement du programme, suivez la valeur attendue du compteur et vérifiez si elle est égale à sa valeur finale lorsque tous les threads créés auront terminé.

Exercice 1.2

Implémentez un système avec plusieurs producteurs et plusieurs consommateurs fonctionnant en parallèle. La terminaison de chaque producteur sera aléatoire. Les consommateurs doivent terminer lorsqu'il n'y a plus de producteurs. Le nombre de producteurs est choisi par `random.randint(1,16)`, de même pour les consommateurs. Les producteurs produisent des couples `(nom, nombre)` où `nom` est le nom du thread et `nombre` est un nombre choisi par `random.randint(1,100)`.

Exercice 1.3

Utilisez le principe « *divide and conquer* » dans l'implémentation parallèle d'une fonction `p_sum(l)` qui renvoie la somme des éléments d'une liste. Le parallélisme sera borné c. à. d. que le résultat sur une liste d'une taille inférieur à un seuil sera renvoyé grâce à la fonction prédéfinie `sum(l)`.

Exercice 1.4

On commence par écrire une version sequentielle de `ind(elt, lst)` qui renvoie une liste d'indicateurs de position d'un élement `elt` dans la liste `lst`. Par exemple `ind(4, [4,2,1,4])` renvoie `[True, False, False, True]`.

- Le but de l'exercice : implémenter une version parallèle de `ind(elt, lst)`.
- L'implémentation utilise une constate qui fixe la taille d'une tranche. Chaque thread se charge d'une tranche et on rassemble les résultats partiels pour forme la liste indicatrice.

Vous devez implémenter

1. une version utilisant `threading`,
2. une version utilisant `concurrent.futures`.

Exercice 1.5

Implémentez un quicksort parallèle. L'implémentation sera testée sur une liste engendrée de façon aléatoire

`[random.randint(1, 1000) for _ in range(10000)]`.

À la fin du tri, on affiche quelques premiers et quelques derniers éléments de la liste triée.

1. Pour une première tentative, on s'appuie sur le module `queue`. Des objets de la classe `Queue` seront utilisés pour récupérer des résultats partiels.
2. On observe que les aspects «moniteur» de la classe `Queue` ne servent pas dans cette implémentation puisque les calculs s'effectuent sur des parties disjointes des listes. Modifiez la solution précédente pour s'affranchir du module `queue`.
3. Ajoutez une collecte des statistiques sur le nombre des threads de votre implémentation actifs en même temps.
4. Pour finir, proposez une solution utilisant `concurrent.futures`.