Programmation avec des threads en Python

Exercice 1 Produit de Matrices

Le calcul du produit $P = A.B$ de deux matrices A et B de dimension respective $(M \times N)$ et $(N \times P)$ est donné par la formule :

$$P_{i,j} = \sum_{1 \leq k \leq N} A_{i,k} * B_{k,j}$$

On remarque que les calculs des $P_{i,j}$ sont indépendants les uns des autres. Ecrire un programme Python qui effectue ce calcul à l'aide de *threads*.

Exercice 2 Puzzle mathématique

On souhaite résoudre le "puzzle" mathématique suivant : étant donnée un entier k et une chaîne de caractères `s`, trouver un entier `n` tel que la valeur de hachage de la paire `(s,n)` est égale à 0 modulo 10^k. En d'autre termes, en utilisant la fonction prédéfinie `hash` de Python, on cherche à trouver un entier `n` tel que :

```
abs(hash((s,n))) % pow(10,k) == 0
```

De part les propriétés de la fonction d'une fonction de hachage, il n'existe pas de méthode autre que la force brute pour résoudre ce problème. Aussi, pour accélérer le calcul, on peut utiliser plusieurs threads qui vont essayer des valeurs de `n` différentes. Par exemple, on peut commencer par utiliser deux threads, le premier essayant des valeurs de `n` *paires*, et l'autre des valeurs *impaires*. Réaliser un programme Python pour résoudre ce puzzle avec ces deux threads. Ce programme prendra en argument la chaine `s` ainsi que l'entier `k`. Le programme devra se terminer dès qu'un résultat sera renvoyé par un des deux threads.

Exercice 3 Puzzle mathématique pour une liste

On souhaite maintenant résoudre ce puzzle pour une liste `t` de chaînes de caractères. Toujours en utilisant deux threads, écrire un programme qui résoud le puzzle pour chaque chaîne du tableau en utilisant les deux threads. La difficulté est ici que le premier thread qui trouve une solution doit arrêter l'autre thread afin que les deux passe au calcul sur la chaîne suivante.

Exercice 4 Barrière de Synchronisation à usage unique

Une *barrière de synchronisation* permet à n threads de s'attendre à un point de leur programme. Une barrière à usage unique n'est utilisée qu'une seule fois par les threads. Implémenter en Python une barrière à usage unique comme un objet de classe `Barriere` avec l'interface suivante :

`b = Barriere(n)` permet de créer une barrière `b` pour `n` threads

Un appel `b.wait()` par un thread le bloque tant que tous les autres threads n'ont pas eux-mêmes réalisés cet appel.

Exercice 5 Barrière de Synchronisation réutilisable

On cherche maintenant à réaliser une barrière de synchronisation qui puisse être réutilisée par les threads dans un calcul qui nécessite plusieurs synchronisation. Étendre votre classe `Barriere` pour cela.

On pourra utiliser le schéma suivant :

```
1  shared int count = P;
2  shared bool sense = TRUE;
3  private bool l_sense = TRUE;

4  barrier:
5          l_sense = not l_sense;
6          Lock();
7             count --;
8             if (count == 0) {
9                     count = P;
10                    sense = l_sense;
11            }
12         Unlock();
13         while (sense != l_sense) ;
```

Mini-projet : Le Jeu de la Vie

Le *jeu de la vie* de Conway représente l'évolution d'une population de cellules contenue dans un tableau bidimensionnel. Chaque case du tableau contient 0 ou 1 cellule et on simule l'évolution de la population en divisant le temps en une suite d'instants et en calculant (suivant des règles décrites plus loin) la population à chaque instant.

Règles d'évolution : Pour savoir l'état d'une case à l'étape $n+1$, on regarde son état et celui de ses 8 voisines à l'instant n.

— Si elle est **vide** et qu'elle a *exactement* 3 cases voisines **occupées**, elle devient **occupée** par une nouvelle cellule. Sinon elle reste vide.

— Si elle est **occupée** et qu'elle a exactement 2 ou 3 cases voisines également **occupées**, la cellule qui occupe la case survit. Sinon le cellule disparaît.

1. Ecrire un programme qui simule l'évolution d'une population de cellules de dimension $n \times n$ à l'aide d'un programme séquentiel. Pour simplifier les calculs, on prendra deux tableaux $(n+2) \times (n+2)$. Le premier contiendra les cellules et le deuxième sera utilisé pour calculer le nombre de cellules voisines occupées. Après l'initialisation de la population de manière aléatoire, la boucle principale du programme consistera à afficher puis à calculer la nouvelle population.
2. On remarque que le calcul de l'état d'une case à l'étape suivante est indépendant du calcul des autres cases. Modifier votre programme afin d'effectuer ces calculs à l'aide de *threads* indépendantes.
3. Plutôt que de créer $n \times n$ *threads* à chaque itération, on veut maintenant créer $n \times n$ *threads* qui calculent *en boucle* l'état de leur case respective. La difficulté consiste à *synchroniser* les *threads* entre elles pour qu'aucune ne commence à calculer l'étape $n+2$ si d'autres n'ont pas encore fini de calculer l'étape $n+1$ (il faudra donc utiliser une *barrière de synchronisation*).

Pour réaliser une version graphique de ce jeu, vous pouvez utiliser la bibliothèque `tkinter` qui permet d'afficher facilement une grille.