THREADS JAVA

Rendu de TD

Les réponses aux exercices sont à rendre sur votre dépôt `github` de nom « M412_Prenom_NOM » (https://github.com/uns-iut-info/), en respectant les consignes suivantes :

- ☐ tous les fichiers seront placés sous le répertoire `M412/TD_THREADS` (en majuscules) ;
- ☐ **dans le répertoire** `M412/TD_THREADS`**, vous devez créer un fichier vide (en respectant la typographie sans accents) :** `M412/TD_THREADS/TD_THREADS_Prenom1_NOM1_et_Prenom2_NOM2`
- ☐ un fichier `README.md` qui décrit votre avancement : exercices terminés, en cours, difficultés éventuelles rencontrées, ...
- ☐ quand c'est demandé, fournir un document texte de nom `exercice_num.md` avec les réponses de l'exercice numéro num ;
- ☐ les fichiers sources Java **commentés avec votre/vos noms en en-tête** doivent utiliser le codage UTF-8 et respecter les noms de l'énoncé ;
- ☐ chaque programme doit proposer dans la fonction `main()` des démonstrations en mode silencieux (aucune entrée/sortie interactive), les tests peuvent être fournis à part sous forme de modules `Junit 5`.
- ☐ Vous pouvez utiliser les squelettes Java fournis sur https://github.com/uns-iut-info/m412-skel-td1.git : `git clone https://github.com/uns-iut-info/m412-skel-td1.git`

---

1 Entrelacement des Threads : PingPong

1.1 Classe PingPong dérivée de Thread

1. Modifier la classe `HelloThread` pour créer la classe `PingPongThread`
2. Ajouter les attributs de classe `String nom` et `int nb` ainsi que le contructeur qui permet de donner un nom et un nombre à l'objet `Thread` crée.
3. La nouvelle méthode `run()` affichera `nb` fois le `nom` passé en argument
4. Lancer en parallèle un thread de nom `ping` et un thread de nom `pong`
5. Observer l'entrelacement des affichages
6. Ralentissez l'affichage en ajoutant une temporisation aléatoire dans l'affichage. Pour générer un délai aléatoire on utilisera simplement `Math.random() * 1000`

1.2 Classe PingPong qui implémente Runnable

Écrire une classe `PingPongRunnable` équivalente en fonctionnalité à `PingPongThread`.

2 SynchronizedCounter

Soit la méthode `SynchronizedCounter()` vue en cours :

1. Modifier la classe `SynchronizedCounterThread` pour accepter en paramètres `nInc` et `nDec` le nombre de fois que l'on invoque `increment()` et `decrement()` sur l'objet de type `SynchronizedCounter`. Ainsi on pourra créer des threads comme :

```
sct1 = new SynchronizedCounterThread(count, 10000, 5000);
Thread t1 = new Thread(sct1);
sct2 = new SynchronizedCounterThread(count, 5000, 10000);
Thread t2 = new Thread(sct2);
```

2. Donner (vérifier) les résultats obtenus pour :
   (a) (count, 10000, 5000) et (count, 10000, 5000)
   (b) (count, 10000, 10000) et (count, 10000, 10000)
   (c) (count, 5000, 10000) et (count, 5000, 10000)
3. Enlever le mot clé `synchronized` à la déclaration de `decrement()` et refaire les calculs précédents. Quels sont les nouveaux résultats ?
4. Utiliser un objet `lock` comme dans le cours (diapo 28) pour écrire les nouvelles classes `LockCounter` et `LockCounterThread` qui permettent d'invoquer `increment()` et `decrement()` sans erreur et sans utiliser la déclaration `synchronized` sur des méthodes (mais sur un objet).
5. Utiliser la classe `AtomicInteger` pour écrire les nouvelles classes `AtomicCounter` et `AtomicCounterThread` qui permettent d'invoquer `increment()` et `decrement()` sans erreur et sans utiliser la déclaration `synchronized`.

3 Somme multithread

Soit un tableau de `n` entiers distincts (avec `n` très grand), on souhaite écrire la méthode multithread `somme` qui calcule la somme des éléments de ce tableau. Le nombre de threads `p` est passé en argument de la ligne de commande.

La classe Java permettant de réaliser se calcul sera nommé `SommeThread.java`. Le calcul d'une somme partielle (sur une partie du tableau allant de l'indice `debut` à l'indice `fin`) pourra être implémenté dans une classe distincte, `SommeIntervalle.java`.

4 Tri Fusion

Dans cet exercice on va écrire un programme de tri de tableaux d'entiers avec la classe `TriFusion`.

- Le nombre de threads `p` est passé en argument de la ligne de commande
- Chaque thread trie une tranche de tableau de taille `tab.length / p`
- Les tranches triées sont fusionnées (interclassées) deux à deux pour produire le résultat final

Bonus : réaliser la fusion en utilisant ici aussi le multithread.

5 Recherche multithread

Soit un tableau de `n` éléments distincts non ordonnés (avec `n` très grand), on souhaite écrire une méthode multithread de recherche d'un élément dans ce tableau. On va modifier la méthode `recherche()` donnée ci-après pour que chaque thread effectue la recherche sur une partie du tableau seulement.

- Le nombre de threads est passé en argument de la ligne de commande

- Dès qu'un thread a trouvé l'élément recherché, on doit stopper les autres. On peut par exemple utiliser une variable globale.

```
1 public int recherche(int[] tab, int x) {
2   for (int i = 0; i < tab.length; i++) {
3     if (x == tab[i]) {
4       return i;
5     }
6   }
7   return -1;
8 }
```