TP : Thread et concurrence en Java

Pour la réalisation de TP, penser à vous référer à l'API Java :
`http://docs.oracle.com/javase/7/docs/api/`

1 Echauffements

1- Écrire un programme qui va lancer 4 thread. Chaque thread affichera son identifiant deux fois (`System.out.println(''Je suis le Thread X'')`) dans une boucle infinie. Que constatez vous ?

2- Créer un programme qui lance plusieurs threads qui vont accéder à une variable statique commune appelée `compteur` sans vérification de l'exclusion mutuelle. Mettre en avant que plusieurs exécutions du programme peuvent founir un résultat différent.
– Quelles sont les solutions que vous connaissez pour éviter ce problème ?
– Proposer une modification suivant l'une de vos propositions.

2 Lecteurs / rédacteurs

1- Codez les différentes stratégies de lecteurs/rédacteurs vues en cours à l'aide de l'objet `Semaphore`.

2- Testez vos programmes pour vérifier que les stratégies sont respectées.

3 Producteurs et consommateurs

1- Ecrire un programme Java mettant en place un mécanisme producteurs/consommateurs avec un buffer contenant un seul élément. Assurer l'exclusion mutuelle.

2- Ecrire un programme Java mettant en place un mécanisme producteurs/consommateurs avec un buffer représenté sous la forme d'un tableau pouvant contenir n éléments. Assurer l'exclusion mutuelle.

4 Calcul parallèle à l'aide de threads

On veut faire la somme de deux matrices carrées A et B contenant `double` de N lignes et autant de colonnes.

La matrice A ne contient que des 1, c'est-à-dire :

$$\forall i, j \in [0, N-1], a_{i,j} = 1$$

La matrice B est définie comme suit :

$$\forall i, j \in [0, N-1], b_{i,j} = i \times N + j$$

Le résultat de l'addition entre A et B est rangé dans une matrice C.

L'objectif est de répartir le calcul de l'addition sur plusieurs threads. Le calcul de chaque élément de la matrice C étant indépendant des autres, dans le cadre d'une architecture parallèle (plusieurs processeurs ou plusieurs cœurs sur un même processeur) le calcul pourra se faire en parallèle.

Les matrices A, B et C sont engendrées par le processus qui va lancer les threads. L'addition se fera en parallèle en lançant 4 threads, chaque thread gèrant un quart de la somme.

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \quad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} \quad C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$$
$$C_i = A_i + B_i \quad \text{pour } i = 1, 2, 3, 4$$

1- Définissez 4 threads, chacun réalisant la somme dans un quart de la matrice.

2- Écrivez le `main()` de votre programme qui devra réaliser les opérations suivantes :

1. Initialiser les matrices A, B et C ;
2. Lancer 4 threads exécutant effectuant le calcul
3. Attendre la fin de l'exécution des threads ;
4. Vérifier que le résultat (la matrice C) est correcte ($\forall i, j \in [0, N-1], c_{i,j} = i \times N + j + 1$).

5 Réalisation d'un sémaphore

Les sémaphores tels que définis par Dijkstra sont maintenant disponibles depuis Java 1.5 [1]. Cet exercice montre comment utiliser les méthodes de base pour créer des outils de plus haut niveau. Il s'agira donc **de ne pas utiliser** la classe `Semaphore` proposée par Java.

On se propose de réaliser les opérations classiques `P` et `V` sur les sémaphores.

Pour l'opération `P` :
– on utilisera la méthode `wait()` pour bloquer les threads.
– en plus du compteur (appelé ici `counter`), on tiendra également à jour le nombre de threads bloqués en utilisant une variable appelée `blocked` qui indique le nombre de threads qui n'ont pas accès à la ressource.
Pour l'opération `V` :
– on utilisera `notify()` pour débloquer l'un des threads bloqués

6 Synchronisation et barrière

L'objectif est d'écrire une application qui illustre la gestion d'un pool de threads. Un pool de threads fonctionne de la façon suivante : n travaux à effectuer sont en attente, ils vont être pris en charge par m threads ($m < n$). L'objectif est de borner le nombre de threads d'une application. Nous utiliserons ici `wait` et `notify` afin d'assurer la synchronisation.

Scénario : Deux types de threads sont utilisés : `Leader` et `Worker`.

1. Les threads du type `Worker`, au nombre de m, attendent qu'il y ait des travaux à effectuer, ces travaux sont au nombre de n. Les valeurs de n et m sont des paramètres donnés sur la ligne de commande.
2. Un thread du type `Leader` initialise le nombre de travaux à faire et débloque tous les threads du type `Worker`.
3. Tant qu'il y a des travaux à effectuer, les threads `Worker` les prennent en charge. Il y a n travaux pour m threads, et n est supérieur à m.

1. `http://download.oracle.com/javase/6/docs/api/java/util/concurrent/Semaphore.html`

4. Lorsque tous les travaux sont pris en charge, les threads `Worker` qui n'ont rien à faire se mettent en attente de nouveaux travaux.

On utilisera deux objets de synchronisation :

1. Une barrière sur laquelle attendent les `Worker` quand ils n'ont rien à faire et qui est levée par le thread `Leader`
2. Un objet qui contient le nombre de travaux à faire sur lequel se bloque le chef en attendant que tous les travaux aient été pris en charge.

7 Problème du barbier endormi

1- Un coiffeur dispose d'une chaise de coiffure et d'une salle d'attente avec un certain nombre de places. Lorsqu'un coiffeur a fini de couper les cheveux d'un client, il le fait sortir et va à la salle d'attente afin de vérifier s'il n'y a pas d'autres clients. S'il trouve un nouveau client, il l'amène à la chaise et lui coupe les cheveux. S'il n'y a pas de nouveau client, il va dormir sur la chaise. Quand un nouveau client arrive, il vérifie si le coiffeur dort. S'il dort, il le réveille et se fait couper les cheveux. Si le coiffeur est déjà occupé avec un autre client, il va à la salle d'attente. S'il y a assez de place, il attend jusqu'à ce que le coiffeur viennet le chercher, sinon il s'en va. On considère que chaque action (couper les cheveux, se déplacer, vérifier les états des autres personnes) prend un certain temps qui dépend de la personne.
– Proposer une solution en Java à ce problème.

8 Problème du dîner des philosophes

1- Cinq philosophes sont assis autour d'une table ronde, chacun a devant lui une assiette de spaghettis, il faut deux fourchettes pour manger les spaghettis. Au cours de sa vie, chaque philosophe pense ou mange. Quand un philosophe a faim, il tente de prendre des fourchettes à droite et à gauche. S'il y parvient il mange un moment, puis pose les fourchettes et se met à penser. L'objectif est que chaque philosophe puisse faire ces deux activités sans jamais être bloqué
– Proposer une solution en Java à ce problème.

9 Panique au Resto U

Le resto U du campus de l'Illberg est submergé le midi par une foule d'étudiants. Certains étudiants attendent patiemment d'être servis alors que certains impatients tentent de bousculer les autres afin d'être servis en premier. Pour instaurer de l'ordre, le gestionnaire du RU a installé un distributeur de tickets. Les tickets sont numérotés de 1 à M. Lorsque le ticket de numéro M est pris, le ticket suivant est le numéro 1. Le comptoir dispose d'un afficheur qui indique le numéro de l'étudiant à servir. L'étudiant qui possède le ticket de même numéro que celui affiché se rapproche du comptoir pour être servi. Lorsque la demande de service de l'étudiant courant est traitée, on sert un étudiant de numéro supérieur (ou 1 si le denier ticket était M). S'il y a trop d'étudiants, on les fait patienter.

1. Programmez votre solution en Java en implémentant les étudiants sous la forme de threads qui doivent obtenir un ticket avant d'être servis. Proposer un affichage en mode texte (ou graphique) qui simule l'arrivée et la gestion des étudiants. Chaque étudiant prendra un temps aléatoire à être servi.
2. Étendre votre solution pour un nombre C de comptoirs possédant chacun un compteur de tickets. Attention, vérifier qu'un étudiant ne soit présent dans la file d'attente d'un seul comptoir à la fois.
3. Proposer des solutions pour répartir la charge (les étudiants) de façon équitable entre les différents comptoirs.
4. Pensez vous que votre solution minimise le temps d'attente des étudiants ? Votre solution empêche t-elle les interblocages et par quels mécanismes ?
5. Si l'on rajoute une échéance pour chaque étudiant, par exemple sur le fait qu'il doit être servi avant

que ses cours ne reprennent, quel est l'impact sur votre système ? Quels mécanismes pourraient être mis en place pour assurer le respect des échéances ? Est-il possible de garantir que tous les étudiants seront à l'heure en cours ? Modifier votre programme pour implémenter vos solutions.

Remarque : Ce sujet a été composé à l'aide de nombreuses ressources disponibles sur la toile et plusieurs exercices sont inspirés d'exercices disponibles. La composition de ce sujet a été pensé dans un objectif purement pédagogique, et les auteurs respectifs gardent l'entière paternité de leur travail. Pour une liste exhaustive des ressources utilisées, se référer au cours.