TP4 : Les moniteurs en JAVA

La création d'une variable partagée entre threads peut provoquer des conflits, à cause de la préemption qui a parfois lieu à des endroits critiques. Il est ainsi nécessaire, pour éviter cela, de définir une *section critique* dans le programme, et d''en protéger l''accès. Java fournit un mécanisme pour permettre l''exclusion mutuelle : le mot clef *synchronized*, qui peut être utilisé comme attribut d′une méthode ou d''une séquence d''instructions.

1.1 Synchronized

Si on a une donnée partagée dans une classe *Interfere*, sa mise à jour en exclusion mutuelle peut être faite de deux façons :

1.1.1 Exclusion mutuelle sur une méthode :

```
class Interfere
{
        private int data = 0 ;
        public synchronized void update ( ) {
        data++;
        }
}
```

Le mot clef *synchronized* ajouté devant la méthode update permet de définir la méthode update comme étant une section critique, et faire ainsi en sorte qu''un seul thread puisse y accéder à la fois.

1.1.2 Exclusion mutuelle sur une séquence d''instructions :

```
class Interfere
{
      private int data = 0 ;
      public void update ( ) {
           synchronized (this) {
           data++;
           }
      }
}
```

Les deux façons de faire sont basées sur l''existence d''une file d''attente par objet crée : la classe Object en déclare une. Donc l''appel d''une méthode *synchronized* ou l''entrée dans une séquence d''instructions *synchronized* utilise cette file d''attente pour bloquer les processus qui n'ont pas accès à la section critique.

1.2 Wait & Notify

La classe Object déclare également deux méthodes pour la synchronisation : *wait* et *notify*. Ces deux méthodes ne doivent être appelées que dans une portion de code *synchronized*, donc quand l''objet est utilisé en exclusion mutuelle.

La méthode *wait* bloque le processus appelant dans la file d'attente de l''objet courant et libère cet objet. Java ne fournit pas des variables conditionnelles. Toutefois, on peut utiliser la file d''attente d''un objet *synchronized* comme variable conditionnelle implicite (une seule par objet).

La méthode *notify* libère un processus bloqué dans la file d''attente de l''objet. Le processus qui l''appelle continue d''avoir l''accès exclusif à l′objet, donc le processus libéré sera exécuté quand il pourra acquérir l''objet. Il s''agit donc d''une sémantique *signale et continue*. Java fournit également une méthode *notifyAll* qui libère tous les processus bloqués dans la file d''attente.

Toutes les méthodes ci-dessus ont une liste vide de paramètres. Un processus qui exécute un code en exclusion mutuelle peut appeler une autre méthode synchronized. Si cette méthode appartient à un autre objet, l'exclusion mutuelle est maintenue pour le premier objet durant l''appel. Ceci peut provoquer des inter-blocages

si une méthode synchronized d″un objet O1 appelle une méthode synchronized d″un autre objet O2 et vice-versa.

1.3 Sleep

Le méthode *Thread.sleep(time)* sert à endormir le thread courant pendant une période de temps *time*. C″est une manière efficace de rendre le processeur disponible pour les autres threads d″une application ou pour d″autres applications qui sont en exécution. La méthode *sleep* prend comme paramètre un entier, qui représente le nombre de millisecondes pendant lequel le thread va s″endormir.

2 Exercices d″Application

2.1 Exercice 1 : Allocation de ressources

Nous nous intéressons au problème d'allocation de ressources banalisées. Nous supposons disposer dans un système de plusieurs exemplaires d'une même ressource, dont un nombre quelconque peut être demandé par un processus à un moment donné. Pour simplifier, nous considérons un seul type de ressource.
Tout processus actif qui demande des exemplaires de cette ressource, doit solliciter les services d'un allocateur via deux procédures :
---- request (m) : pour demander l'allocation de m exemplaires de la ressource
---- release (m) : pour libérer les m exemplaires de la ressource
L'allocateur maintient une variable critique qui compte le nombre d'exemplaires disponibles afin de satisfaire éventuellement de nouvelles demandes.

```
Contexte commun : entier NbRessDisponibles :=Max ;
Comportement d'un processus :
While (true) {
        request(m) ;
        utiliser les m ressources
        release(m) ;
}
Procédure request (entier m) {
Tant que (NbRessDisponibles<m) bloquer le processus appelant
        ; NbRessDisponibles = NbRessDisponibles –m ;
        // allocation des m ressources
}
Procédure release(entier m) {
        // libération des m ressources
        NbRessDisponibles = NbRessDisponibles +m
        ; Réveiller des processus qui seraient bloqués
}
```

Notons que dans cette solution, plusieurs processus demandeurs peuvent être servis simultanément s'il y a suffisamment de ressources. De plus, la procédure *release* doit réveiller tous les processus en attente pour trouver ceux pour lesquels les demandes peuvent être satisfaites. Traduire cet algorithme en Java en utilisant les moniteurs.

2.2 Exercice 2 : Compteur

On se propose dans cet exercice de créer un compteur en Java en utilisant les moniteurs. Ce compteur sera composé de deux threads, ThreadPair et ThreadImpair. ThreadPair va afficher les nombres pairs, et ThreadImpair les nombres impairs. Le but est de faire en sorte que l″affichage soit :
0 1 2 3 4 5 6 7 ....
Ainsi, ThreadPair commencera par afficher 0, puis ThreadImpair affichera 1, puis ThreadPair 2, etc...

Rq1 : Vous créerez pour cette fin une classe qui s'appelle SynchroClass, qui va contenir une variable entière *tour*. Cette variable représentera prendra respectivement la valeur 0 si c″est le tour de ThreadPair de s'exécuter, et 1 si c″est le tour de ThreadImpair.
Rq 2: Pour mieux visualiser le résultat sur votre console, utiliser la méthode *sleep* pour l″un des threads.