TD 5 – Le partage de ressources

1. Le dîner des philosophes :

- Cinq philosophes sont au restaurant chinois assis à une table circulaire. Il y a seulement 5 baguettes sur la table, une entre chaque assiette. Or pour manger le riz il faut deux baguettes. Ainsi pour manger un philosophe peut prendre une baguette à sa gauche, une autre à sa droite, manger pendant un certain temps puis les reposer. Un de ses voisins peut alors manger, et ainsi de suite. Un même philosophe mange plusieurs fois.
- Créer une classe `Diner` qui instancie 5 variables de la classe `Philosophes` qui sont des threads modélisant l'énoncé ci-dessus. Remarque : il faut également modéliser les baguettes, qui sont des objets partagés entre les threads. Vous pouvez partir avec une partie de la solution (il suffit alors d'écrire la méthode run dans la classe `Philosophe`) :
- Dans la question précédente un cas de blocage est possible : chaque philosophe prend sa baguette droite puis attend indéfiniment d'avoir la gauche. Mettre en évidence ce cas de blocage (par exemple en rajoutant un délai de 1 seconde entre la prise de la baguette droite et la prise de la gauche).
- Une méthode fréquente pour éviter les cas de blocage est de mettre des priorités. On numérote les baguettes de 1 à 5. Pour manger un philosophe prend alors toujours sa baguette de plus petit nombre plus celle de plus grand nombre. Implémenter cette nouvelle solution.

Remarques :
Deadlock (étreinte fatale): les processus s'attendent mutuellement.

Exemple de deadlock des philosophes:

Programme d'un philosophe:

```
loop {
  ramasse baguette à gauche;
  ramasse baguette à droite;
  mange;
  dépose les baguettes;
  pense;
}
```

Deadlock: si chaque philosophe éxécute simultanément ce programme, chacun ramasse la baguette à sa gauche et ensuite ils s'attendent mutuellement pour pouvoir prendre celle à droite.
Note: baguette = une ressource partagée.

Les deadlocks arrivent souvent quand on a une chaîne circulaire de dépendences où un processus attend une ressource (ou une réponse) d'un autre processus dans la chaîne.

Livelock (étreinte active): le calcul ne progresse pas.
Exemple avec le dîner des philosophes:

Programme d'un philosophe:

```
loop {
  ramasse baguette à gauche;
  si baguette à droite est disponible
     ramasse baguette à droite;
  sinon
     relâche baguette à gauche et recommence la boucle;
  mange;
  dépose les baguettes;
  pense;
}
```

•On peut alors avoir une boucle infinie ou les processus sont tous activés à tour de rôle, mais sans que le "calcul" progresse réellement. Par exemple, chaque philosophe fait:

ramasse baguette à gauche;

relâche baguette à gauche;

ramasse baguette à gauche;

relâche baguette à gauche;

ramasse baguette à gauche;

relâche baguette à gauche;

...

Principe d'équité (ou de progrès fini): on veut trouver une solution telle qu'un processus qui veut rouler sera capable de rouler, dans un temps fini: un processus qui veut rouler ne peut pas être bloqué indéfiniment, e.g. chaque philosophe mangera éventuellement.
Solution inéquitable mais simple: deux des philosophes (non-adjacents) gardent le contrôle total de leur baguettes, peuvent donc manger et philosopher, alors que les autres meurent de faim ...

Comment prévenir l'étreinte fatale.
Considérons le cas plus simple de 2 philosophes A et B avec 2 ressources R et S (les 2 baguettes). On peut avoir étreinte fatale si un philosophe prend les baguettes dans l'ordre RS alors que l'autre les prend dans l'ordre SR. On peut l'éviter en forçant tous les processus à demander les ressources dans le même ordre. Cet approche fonctionne pour n'importe quelle quantité de ressources (évite les chaînes circulaires de dépendances), mais elle n'est pas toujours possible (cette contrainte peut être trop forte).

Section critique: section de code traitée comme un évènement atomique.
Exemple en Java pour solutionner le dîner des philosophes:

```
while (true) {
  synchronized (verrouPourRamasser) { // section critique
     if (baguettes_disponibles()) {
       ramasse_gauche();
       ramasse_droite();
     }
     else continue; // saute les étapes suivantes et recommence la boucle
  }
  mange();
  deposeLesDeuxBaguettes();
  pense();
}
```

où le champs `verrouPourRamasser` est un objet visible par tous les philosophes (par exemple un champs de classe ou bien un champs associé à un objet "Table", autour de laquelle sont assis les philosophes). Il sert à garantir qu'un seul philosophe de la table à la fois essaie de ramasser les baguettes.
Notez: si la section critique englobait l'appel à mange, ca ne serait pas très bon car un seul philosophe à la fois pourrait manger.