Algorithmique parallèle et distribuée

Synchronisation de ressources partagées

Pour commencer tranquillement

Exercice 1
A partir de la feuille de TP1, reprendre les deux exercices impliquants des ressources partagées :
- le compteur partagé ;
- l'affichage d'une chaine commune.

Nous avons pu remarquer que, lorsqu'ils sont programmés naïvement et sans synchronisation des variables partagées, ces deux programmes ne produisent pas les résultats attendus. Les modifier de façon à ce que, d'une part, le résultat final soit bien celui escompté, et d'autre part à ce que la méthode main attende la terminaison de tous les threads déclenchés avant d'afficher quoique ce soit et/ou se terminer.

Un problème de rendez-vous

Exercice 2
Plusieurs processus parallèles ont un point de rendez-vous : un processus arrivant au point de rendez-vous se met en attente s'il manque au moins un autre processus. Le dernier arrivé réveillera les processus bloqués.

Voici un exemple de programme, rédigé en pseudo-code, réalisant ce rendez vous :

```
sémaphore mutex = 1, s= 0;
entier NbArrivés = 0; /* nbre de processus arrivés au rendez-vous */

Procédure RDV
Début
  P(mutex);
  NbArrivés = NbArrivés + 1;
  Si (NBArrivés < N) Alors /* non tous arrivés */
    V(mutex); /* on libère mutex et */
    P(s); /* on se bloque */
  Sinon
    V(mutex); /* le dernier arrivé libère mutex et */
```

```
      Pour i = 1 à N-1 Faire V(s); /* réveille les N-1 bloqués,
                                              dans l'ordre d'arrivée */
    Finsi
  Fin
```

Traduire cet algorithme en java et en utilisant le concept de sémaphore détaillé en cours :

- une classe *RendezVous* possédant un constructeur et une méthode *rdv()* telle que spécifiée (*Thread.currentThread().getName()* fournit le nom d'un thread) ;
- une classe *Processus* héritant de la classe *Thread* :
  - chaque processus reçoit le rendez-vous par le constructeur ;
  - dans le méthode *run()*, le processus dort de façon aléatoire (cela permet de simuler un traitement), puis rejoint le point de rendez-vous en appelant la méthode *rdv()* ;
  - lorsque tout le monde est au rendez vous, chaque thread se présente et termine son exécution.

Le problème producteurs/consommateurs

Exercice 3
Voici une implantation incorrecte du problème des producteurs/consommateurs :

```
class Q {
  int n;

  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }

  synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
  }
}


class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;

    while(true) {
    q.put(i++);
```

```
    }
  }
}


class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}


class PC {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
  }
}
```

Ce code permet de lever les soucis d'accès concurrent entre prodicteuurs et consommateurs sur les ressources critiques. Cependant, si vous exécutez ce code, vous remarquerez que :

- le producteur peut déborder le consommateur ;
- le consommateur peut consommer plusieurs fois la même valeur.

Ce qu'il faut faire :

- ne modifiez que la classe *Q*, pas les autres ;
- utilisez les méthodes *wait()* et *notify()* pour envoyer des signaux dans les deux directions et résoudre les soucis présentés précésemment.

Les vases communicants

Exercice 4

```
public class Vases {
  private int[] capacité = new int[2];
  private int[] quantité = new int[2];
  private int total;

  public Vases(int c1, int q1, int c2, int q2) {
    assert(c1 > 0 && q1 >= 0 && q1 < c1);
    assert(c2 > 0 && q2 >= 0 && q2 < c2);
    capacité[0] = c1; quantité[0] = q1;
    capacité[1] = c2; quantité[1] = q2;
    total = q1 + q2;
```

```
    }

    public String toString() {
      return "[" + quantité[0] + ", " + quantité[1] + "]";
    }



// Transfert q centilitres du 1er vers le 2ème vase.
public void transfert12(int q) {
  transfert(0, 1, q);
}

// Transfert q centilitres du 2ème vers le 1er vase.
public void transfert21(int q) {
  transfert(1, 0, q);
}


    // Transfert q centilitres du vase s vers le vase d. Seule la
    // quantité maximale est transférée en fonction de la quantité
    // disponible dans s et de la capacité de d
    public void transfert(int s, int d, int q) {
      assert(q >= 0);
      if (q > quantité[s]) q = quantité[s];
      if (quantité[d] + q > capacité[d]) q = capacité[d] - quantité[d];
      quantité[s] -= q;
      quantité[d] += q;
      System.out.println("transf. " + s + " -> "+ d + "  de " + q + " cc " + this);
      assert(total == quantité[0] + quantité[1]);
    }
  }
```

1. Ecrire un programme :
   - créant un objet de type Vases ;
   - lançant NBTHREADS threads.
2. Chaque thread devra réaliser
   - NBTRANSFERTS transferts dans l'objet créé dans le programme
   - le sens du transfert ainsi que la quantité transférée seront choisis aléatoirement :
     - `Random r = new Random();`
       crée un générateur de nb aléatoire ;
     - `r.nextBoolean() ;`
       retourne un booléen choisi aléatoirement ;
     - `int r.nextInt(int max) ;`
       retourne un entier choisi aléatoirement entre 0 et max
   - Après chaque transfert, le thread s'endormira pendant 10 ms

```
try {
      Thread.sleep(10);
    } catch (InterruptedException e) {}
```