Polytech Marseille

# TD 1 - Rappels
# Threads, synchronisation, sockets (illustration en Java)

Informatique - Formation par alternance HUGo - 5A

Les exercices sont à réaliser **sur feuille** (pas sur machine).

Si besoin, relisez les parties "synthèse du cours" des TD de programmation distribuée de l'année universitaire précédente (voir les annexes avec les versions "express" ou les sujets originaux des TD).

## Exercice 1. Diner des philosophes (sémaphores)

Le problème du diner des philosophes est un grand classique (Djikstra 1970). Cinq philosophes sont réunis pour mener à bien deux activités majeures : penser et manger. Chaque philosophe pense pendant un temps aléatoire, mange (si possible) pendant un temps aléatoire puis se remet à penser. Lorsqu'un philosophe demande à manger, une assiette de spaghettis l'attend. Les cinq assiettes sont disposées autour d'une table ronde comme le montre la figure 1.

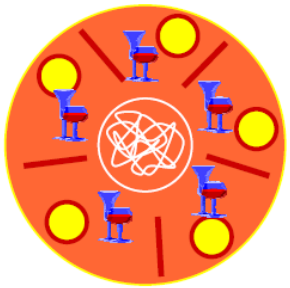

FIGURE 1 – Table du diner des philosophes.

Pour qu'un philosophe puisse manger ses spaghettis, 2 fourchettes sont nécessaires : la sienne (celle de droite) et celle de son voisin de gauche. Naturellement, si l'un de ses voisins ou les deux voisins sont déjà en train de manger, le philosophe ne peut pas manger et doit patienter que l'une ou les deux fourchettes occupées se libèrent. La solution à ce problème doit être la plus optimisée possible : un philosophe voulant mais ne pouvant pas manger doit attendre son tour sans consommer de temps CPU inutilement.

Chaque philosophe effectue plusieurs fois le cycle : PENSE, DEMANDE A MANGER, MANGE.

On doit naturellement constater que deux philosophes voisins ne mangent jamais en même temps et qu'il n'y a pas d'interblocage ni de famine [1].

Proposez une solution (conception sur feuille). Chaque philosophe correspondra à un thread.

1. Le terme "famine" prend ici tout son sens. Il n'y a pas de famine si chaque philosophe voulant manger, mange à un moment donné.

## Exercice 2. Producteur-Consommateur (moniteurs)

Cet exercice aborde le problème du "producteur-consommateur" en utilisant la notion de moniteur. On dispose d'un buffer pouvant des valeurs. Des producteurs veulent déposer des valeurs dans le buffer. Des consommateurs veulent prélever des valeurs se trouvant dans le buffer.

Voici les 4 classes utilisées :

- `BufferCirc` : classe englobant un tableau appelé "tampon" (le buffer) qui contient des `Object`, "taille" correspond à la taille de `tampon`, `prem` est l'indice de l'objet à prélever, `der` est l'indice où déposer l'objet, et `nbObj` est le nombre d'objets dans le `tampon`.
Initialement, `prem` = 0, `der` = 0 et `nbObj` = 0.
Exemple : un buffer circulaire de taille 16 est ici un tableau de `Object` (16 cases, indice de 0 à 15). Il est dit circulaire car après l'indice 15 retour à l'indice 0 (voir plus loin dans le code de la classe `BufferCirc`, par exemple, `der = (der + 1) % taille;`).

- `Producteur` : runnable définissant la tâche effectuée par un producteur.

- `Consommateur` : runnable définissant la tâche effectuée par un consommateur.

- `Principal` : programme principal "main" prenant 3 arguments (la taille du buffer, le nombre de producteurs et le nombre de consommateurs).

Questions

1. Quels sont ici les **problèmes** liés à la concurrence ?
2. Comment résoudre ces problèmes ? (utilisez la notion de **moniteur**)
   Complétez la classe `BufferCirc` en utilisant les moniteurs Java (`synchronized`, méthodes `wait()` et `notifyAll()`). Les autres classes ne sont pas à modifier.
3. Pourquoi utiliser `while(...)` à la place de `if(...)` pour faire le `wait()` ?
4. Pourquoi utiliser `notifyAll()` à la place de `notify()` ?
5. Proposez une solution aux problèmes évoqués dans la première question **en n'utilisant que des sémaphores**.
6. Il existe d'autres mécanismes de synchronisation. Citez-en quelques uns.

Voici le contenu des classes initiales :

Classe "BufferCirc" (à compléter) :

```
public class BufferCirc {

 private Object[] tampon; // tableau d'objects (c'est le contenu du buffer)
 private int taille; // taille max du buffer
                     // (nombre d'emplacements du tableau "tampon")

 private int prem; // indice de l'emplacement de l'object a prelever
 private int der; // indice de l'emplacement ou deposer un object
 private int nbObj; // nombre d'objets presents dans le "tampon" (le buffer)

 public BufferCirc (int t) {
   taille = t;
   tampon = new Object[taille];
   prem = 0;
   der = 0;
   nbObj = 0;
 }
```

```
 public void depose(Object o) { // a completer
   // a completer
   tampon[der] = o;
   der = (der + 1) % taille;
   nbObj = nbObj + 1;
   System.out.println (Thread.currentThread().getName() + "␣a␣depose␣" + (Integer)o);
 }

 public Object preleve() { // a completer
   // a completer
   Object o = tampon[prem];
   tampon[prem] = null;
   prem = (prem + 1) % taille;
   nbObj = nbObj - 1;
   System.out.println (Thread.currentThread().getName() + "␣a␣preleve␣" + (Integer)o);
   return (o);
 }

}
```

Classe "Producteur" (à ne pas modifier) :

```
public class Producteur implements Runnable {

 private BufferCirc buffer;
 private int val;

 public Producteur(BufferCirc b) {
   buffer = b;
 }

 public void run() {
   while (true) {
      buffer.depose(new Integer(val));
      System.out.println (Thread.currentThread().getName() +    "␣a␣depose␣" + val);
      val++;
   }
 }
}
```

Classe "Consommateur" (à ne pas modifier) :

```
public class Consommateur implements Runnable {

 private BufferCirc buffer;

 public Consommateur(BufferCirc b) {
   buffer = b;
 }

 public void run() {
   Integer val;
   while (true) {
      val = (Integer) buffer.preleve();
      System.out.println (Thread.currentThread().getName() + "␣a␣preleve␣" + val);
      try {
         Thread.sleep((int)(Math.random()*1000));
      }
      catch (InterruptedException e) {}
   }
 }
}
```

Classe "Principal" (à ne pas modifier) :

```
import java.util.ArrayList;

public class Principal {

 public static void main (String[] args) {
   if(args.length<3) {
      System.err.println(
          "Usage: java Principal tailleBuffer nbreProducteurs nbreConsommateurs\n");
   }
   else {
      int tailleBuffer = 0;
      int nbProd = 0;
      int nbCons = 0;
      BufferCirc b = null;
      Producteur p = null;
      Consommateur c = null;
      Thread t = null;
      ArrayList<Thread> listeProd = null;
      ArrayList<Thread> listeCons = null;

      try {
         tailleBuffer = Integer.parseInt(args[0]);
         nbProd = Integer.parseInt(args[1]);
         nbCons = Integer.parseInt(args[2]);

         listeProd = new ArrayList<Thread>();
         listeCons = new ArrayList<Thread>();

         b = new BufferCirc(tailleBuffer);

         for(int i=0; i<nbProd; i++) {
            p = new Producteur(b);
            t = new Thread(p);
            t.setName("P"+(i+1));
            listeProd.add(t);
         }

         for(int i=0; i<nbCons; i++) {
            c = new Consommateur(b);
            t = new Thread(c);
            t.setName("C"+(i+1));
            listeCons.add(t);
         }

         for(Thread th : listeProd) {
            th.start();
         }

         for(Thread th : listeCons) {
            th.start();
         }
      }
      catch(NumberFormatException e) {
            System.err.println(e.getMessage());
            e.printStackTrace();
      }
   }
 }
}
```

## Exercice 3. Producteur-Consommateur sur plusieurs machines (sockets TCP)

Dans cet exercice, les producteurs et les consommateurs sont sur différentes machines et le buffer circulaire est une machine particulière.

Imaginez le fonctionnement de ce "producteur-consommateur en réseau" et concevez une solution. La communication se fera via le protocole de transport TCP.

## Exercice 4. Calcul de $\pi$ sur plusieurs machines (sockets UDP)

L'objectif de cet exercice est de concevoir une version parallèle du calcul de la valeur approchée de $\pi$ en utilisant plusieurs machines.

La formule suivante permet d'approcher la valeur de $\pi$ :

$$\pi \quad \cong \quad \sum_{i=1}^{n} \frac{1}{n} \frac{4}{1 + \left((i - \frac{1}{2})\frac{1}{n}\right)^2}$$

Dans cette formule $n$ représente la précision désirée. Plus $n$ est grand et plus la valeur de $\pi$ est précise.

Paralléliser un algorithme consiste à le découper en petits problèmes indépendants qui pourront donc être traités de manière isolée. Cela correspond, dans notre contexte, à réécrire la formule de façon à faire apparaître des parties pouvant être calculées de façon indépendante (c'est le principe de la décomposition fonctionnelle).

Nous pouvons réécrire la formule comme ceci :

$$\pi \approx \sum_{k=1}^{N} \sum_{i=1+(k-1)\frac{n}{N}}^{k\frac{n}{N}} \frac{1}{n} \times \frac{4}{1 + \left((i - \frac{1}{2})\frac{1}{n}\right)^2}$$

Le calcul est ainsi divisé en $N$ plusieurs sommes partielles, chacune intervenant sur un domaine de largeur $n/N$.

$N$ correspond naturellement au nombre de threads utilisés.

Enfin, nous utilisons le paradigme de parallélisation "maître/esclave". Il y a un maitre (i.e., un chef d'orchestre) et plusieurs esclaves ($N$ dans la formule précédente).

Chaque somme partielle sera effectuée par un esclave. La somme générale (la somme de gauche, avec $k$ de 1 à $N$) sera réalisée par le maître.

Concevez une version qui permettra de réaliser le calcul sur différentes machines (une machine pour le maître et une machine par esclave). La communication se fera via le protocole de transport UDP.