Mémoire partagée et sémaphores en java

Frédéric Guinand

Département informatique / IUT du Havre

Avoir plusieurs threads pose des problèmes et des questions

Synchronisation

- comment les différents threads peuvent se synchroniser ?
- comment plusieurs threads peuvent partager des données ?

  les données risquent de perdre leur intégrité si deux threads les manipuler en "même temps"

Problématique

- Si chaque thread manipule des données qui lui sont propres alors pas de problème !
- Si plusieurs threads manipulent des données communes alors :
  - Problème :
    - Espace de travail commun, pas de mémoire privée pour chaque thread
    - Accès simultané à une même ressource
    - Les données risquent leur intégrité si deux threads se retrouvent à les manipuler en même temps
  - Exemple :
    - On considère une bonbonnière et un ensemble d'enfants qui peuvent avoir accès à la bonbonnière sans restriction

Exemple

- on considère une seule bonbonnière (objet partagé)
- les enfants sont des threads
- un enfant ne prend qu'un bonbon à la fois dans la bonbonnière
- à chaque demande de bonbon, la bonbonnière :
  1. délivre un bonbon numéroté à l'enfant
  2. décrémente le nombre de bonbons qui restent

```
public class Principale {
        instancie un objet Bonbonniere en
                passant le nombre initial de bonbons en paramètre
                au constructeur de Bonbonniere
        instancie k objets Enfant (chaque Enfant est un thread) en
                passant à leur constructeur la référence à Bonbonniere
                et un identifiant entier
}
```

Exemple

```
class Bonbonniere {
        int nbBonbons ;
        public Bonbonniere(...) { ... }
        public int attrapeBonbon() {
                int bonbon = 0 ;
                if ( nbBonbons > 0) {
                    bonbon = nbBonbons ; // instruction A
                    nbBonbons - - ; // instruction B
                }
                return bonbon ;
        }
}
class Enfant extends Thread {
        public Enfant(...) { ... }
        public void run() {
                int bonbon = bonbonniere.attrapeBonbon() ;
                while(bonbon  != 0) {
                    S.out.P("enfant :"+id+" bonbon : "+bonbon) ;
                    bonbon = bonbonniere.attrapeBonbon() ;
                    nb++ ;
                }
                S.out.P("nb bonbons mangés  : "+nb) ;
        }
}
```

- Exécutez ce code.
- Que constatez-vous ?
- Modification : vous ajoutez maintenant entre les instructions A et B du code de la Bonbonnière un bout de code qui simule un traitement un peu long. Que constatez-vous ?

Constatations

1. les messages sont affichés dans un ordre quelconque
2. tous les enfants ne mangent pas le même nombre de bonbons
3. certains bonbons sont récupérés plusieurs fois, d'autres n'apparaissent jamais
4. le nombre de bonbons cosommés est différent du nombre initial de bonbons dans la bonbonnière !!

Diagnostic

- les points 1 et 2 relèvent de l'indéterminisme et correspondent à un comportement normal d'un système réparti/distribué, par contre...
- les points 3 et 4 sont les symptômes d'un problème, dans la logique de l'application, chaque bonbon doit être consommé une fois et une seule

Problèmes

1. certains bonbons sont récupérés plusieurs fois, d'autres n'apparaissent jamais
2. le nombre de bonbons cosommés est différent du nombre initial de bonbons dans la bonbonnière !!

Solution

- garantir l'accès exclusif à un objet pendant l'exécution de la méthode

  $\Rightarrow$ `synchronized` permet de gérer les concurrences d'accès

```
class Bonbonniere {
        int nbBonbons ;
        public Bonbonniere(...) { ... }
        public synchronized int attrapeBonbon() {
                int bonbon = 0 ;
                if ( nbBonbons > 0) {
                    bonbon = nbBonbons ;
                    ... /* long calcul */
                    nbBonbons - - ;
                }
                return bonbon ;
        }
}
```

Synchronisation (1/2)

Lors de l'appel à une méthode `synchronized`

- l'objet est-il accessible (non-verrouillé) ?
  - Si oui, l'objet est verrouillé et le thread exécute la méthode
  - Si non, le thread doit attendre la levée du verrou (c'est-à-dire que l'exécution de la méthode (verrouillée) par un autre thread soit terminée)
- On appelle section critique :
  - Une méthode : déclaration précédée de `synchronized`
  - Une instruction (ou un bloc) : précédée de `synchronized`
  - Un objet : le déclarer `synchronized`

Synchronisation (2/2)

- Pour gérer la concurrence d'accès à une méthode :
  - Si un thread exécute cette méthode sur un objet, un autre thread ne peut pas l'exécuter pour le même objet
  - En revanche, il peut exécuter cette méthode pour un autre objet

```
public synchronized void maMethode() {...}
```

- Pour Contrôler l'accès à un objet :

```
public void maMethode() { ...
     synchronized(objet) {
          objet.methode() ;
      }
 }
```

  - l'accès à l'objet passé en paramètre de `synchronized(Object)` est réservé à un unique thread

Exercice

- testez ce code, allongez les suites de 0 et de 1 jusqu'à observer des coupures puis
- modifiez-le pour que l'affichage des chaînes de 0 ou de 1 ne soit pas interrompu.

```
class Affichage {
  public void afficher(String t) {
       for (int i=0 ; i<t.length() ; i++)
            System.out.print(t.charAt(i)) ;
       System.out.println() ;
  }
}
public class TPrint extends Thread {
  static Affichage mAff = new Affichage() ;
  String txt ;
  public TPrint(String t) {txt = t ;}
  public void run() {
       for (int j=0 ; j<20 ; j++)
            mAff.afficher(txt) ;
  }
  public static void main(String args[]) {
      new TPrint("00000000000000000000000000").start() ;
      new TPrint("11111111111111111111111111").start() ;
  }
}
```

Limites de *synchronized*

- l'utilisation de *synchronized* ne garantie pas l'ordre dans lequel les threads accèdent à la ressource partagée
- c'est l'ordonnanceur qui décide quel thread sera la prochain à exécuter du code

  $\Longrightarrow$ si l'ordre d'accès est important il faut un mécanisme supplémentaire
- par exemple si deux threads doivent exécuter un code à tour de rôle, *synchronized* ne permet pas à lui tout seul de régler le problème.

Exemple illustrant les limites de *synchronized*

- on considère deux threads qui doivent effectuer une action à tour de rôle, comme les échanges au tennis ou au ping-pong
- ils ont en partage la balle
- essayez de coder cet exemple avec la Balle qui est l'objet partagé, et deux Joueurs (deux threads)
- comment gérez-vous l'alternance des actions des deux threads sur la Balle ?
- indication : la balle doit garder en mémoire l'identifiant du dernier thread qui l'a accédée

Exemple

```
class Joueur extends Thread {

    int id;
    int score = 0;
    Random alea;
    Balle balle;

    public Joueur(int id, Balle b) {
        this.id = id; balle = b;
    }

    public void run() {
        while(balle.tape(id)) {
            score++;
        }
        System.out.println("Joueur "+id+" a terminé après "+score+" échanges");
    }
}
```

Exemple

```
class Balle {

    boolean encours = true;
    Random alea;
    int echange = 0;

    public Balle() { alea = new Random(System.currentTimeMillis()); }

    public synchronized boolean tape(int idJoueur) {

        if(encours) {
            if(alea.nextInt(100) > 20) { // echange continu
                System.out.println("Joueur "+idJoueur+" a réussi à taper la balle");
                echange++;
            } else {
                encours = false;
                System.out.println("Joueur "+idJoueur+" a raté la balle");
                System.out.println("longueur de l'échange "+echange);
            }
        }
        return encours;
    }
}
```

Exemple (traces d'exécution)

```
public class PingPong {

    Joueur j1, j0;

    public static void main(String[] args) {
        new PingPong();
    }

    public PingPong() {
        Balle b = new Balle();
        j0 = new Joueur(0,b);
        j1 = new Joueur(1,b);
        j0.start();
        j1.start();
    }
}
```

```
Joueur 0 a réussi à taper la balle
Joueur 0 a réussi à taper la balle
Joueur 0 a réussi à taper la balle
Joueur 0 a réussi à taper la balle
Joueur 0 a raté la balle
longueur de l'échange 4
Joueur 0 a terminé après 4 échanges
Joueur 1 a terminé après 0 échanges
```

pas vraiment des échanges...

Ordre d'exécution des threads (via un objet partagé)

- pour résoudre ce problème, une solution consiste à mettre en attente le dernier thread qui a eut accès à la ressource partagée, ce qui peut être réalisé par la primitive *wait()*

  $\rightarrow$ *try { wait() ; } catch(InterruptedException ie) {}*
- il faut également un mécanisme de réveil du thread en attente lorsque l'accès a été libéré, c'est réalisé par la primitive *notify()* (le premier thread en attente) ou *notifyAll()* (tous les threads en attente sont prévenus)

Méthodes wait() / notify() / notifyAll()

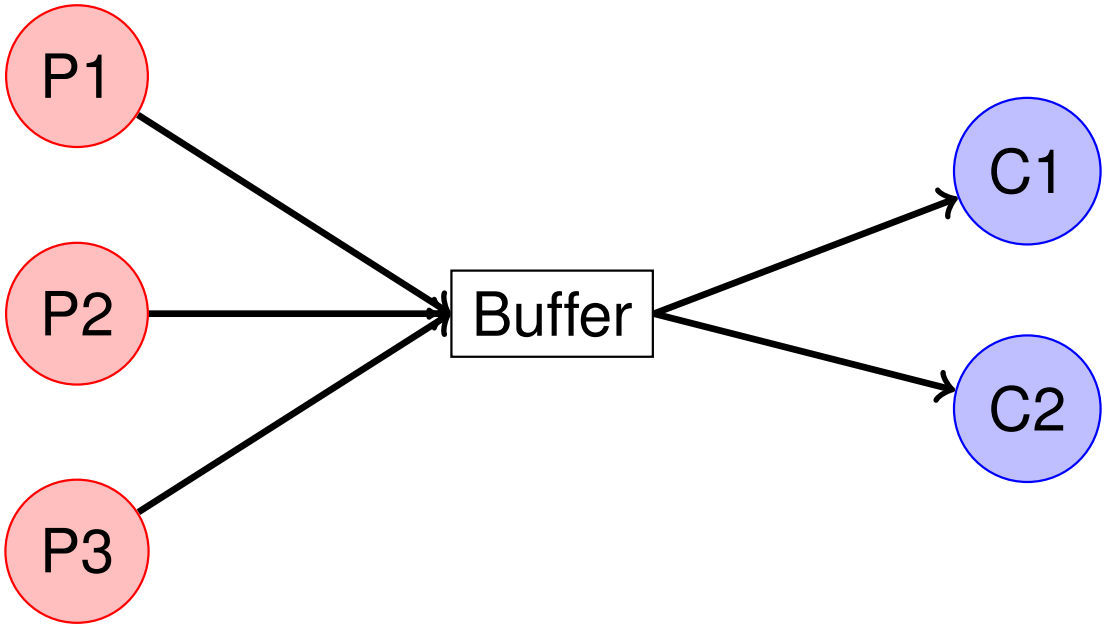
- `wait()` :
  - Doit être appelée depuis l'intérieur d'une méthode ou d'un bloc synchronized
  - Attend l'arrivée d'une condition sur l'objet sur lequel elle s'applique
  - Le thread en cours d'exécution est bloqué
  - Pendant que le thread attend, le verrou est relâché
  - Il existe aussi `wait(long temps)` qui termine l'attente après temps millisecondes, si cela n'est pas fait auparavant
- `notify()` / `notifyAll()` :
  - `notify()` : libère le thread dans l'état wait
    - Si plusieurs threads sont dans l'attente imposée, `notify()` débloque celui qui attend le plus longtemps
  - `notifyAll()`, fait la même chose mais pour tous les threads en attente

```
class Balle {

    boolean encours = true;
    int joueurencours = 0;
    Random alea;
    int echange = 0;

    public Balle() {
        alea = new Random(System.currentTimeMillis());
        if(!alea.nextBoolean()) joueurencours = 1;
        System.out.println("engagement pour le joueur "+joueurencours);
    }

    public synchronized boolean tape(int idJoueur) {

        while(idJoueur != joueurencours) { try { wait(); } catch(InterruptedException ie) {} }
        if(encours) {
            if(alea.nextInt(100) > 20) { // echange continu
                System.out.println("Joueur "+joueurencours+" a réussi à taper la balle");
                echange++;
            } else {
                encours = false;
                System.out.println("Joueur "+joueurencours+" a raté la balle");
                System.out.println("longueur de l'échange "+echange);
            }
        }
        joueurencours = (joueurencours+1)%2;
        notify();
        return encours;
    }
}
```

Producteurs / Consommateurs

- ceci nous amène à un problème très fréquent en informatique,
- problème que l'on retrouve des couches logicielles basses (système d'exploitation) jusqu'au niveau applicatif :

  $\rightarrow$ le problème Producteur/Consommateur, aussi connu sous le nom Lecteur/Rédacteur.



Producteur/Consommateur

Principe

- un ensemble de threads accèdent à un espace partagé pour y déposer des items

  $\rightarrow$ les producteurs
- d'autres threads y accèdent pour retirer ces items afin de les traiter

  $\rightarrow$ les consommateurs
- un problème se pose lorsque :
  1. des consommateurs tentent de retirer des items de la zone de stockage et que celle-ci est vide
  2. des producteurs tentent d'ajouter des items à une zone ayant atteint sa capacité maximale de stockage

Exercice

- Les producteurs/consommateurs partagent un buffer de capacité égale à $N > 0$.
- Les producteurs et les consommateurs doivent se synchroniser sur le buffer.
- Un producteur doit attendre que le buffer ne soit pas plein pour ajouter un item dans le buffer.
- Un consommateur attend que le buffer ne soit pas vide pour consommer un item.

- En vous inspirant de l'exemple du ping-pong et en utilisant le mot clef *synchronized* ainsi que les méthodes `wait()` et `notify()` (et éventuellement `notifyAll()`) proposez une solution
- la solution sera composée de quatre classes : Buffer, Producteur, Consommateur et Lanceur (qui contient le main).
- les classes Producteur et Consommateur étendent la classe Thread

Tous les problèmes d'accès à des ressources partagées sont-ils résolus ? NON

Exemple

- considérons deux équipes de deux personnes : $\{A_1, A_2\}$ et $\{B_1, B_2\}$,
- pour se contacter, chaque équipe doit disposer de deux téléphones, or il n'y a que deux téléphones
- hypothèse, lorsqu'une personne a accès à un téléphone, elle ne le lâche qu'à l'issue de la communication avec son partenaire,
- que se passe-t-il si A_1 et B_2 ont accès chacun à un téléphone ?

on se trouve dans une situation d'interblocage

Producteur et Consommateur

- Producteur.java $\bigcup$ Consommateur.java $\rightarrow$ ProdConso.java
- Buffer.java reste inchangé
- Dans la méthode run(), le fait de produire ou de consommer est choisi aléatoirement à chaque tour de boucle
- au moment de l'instantiation des ProdConsos, 3 paramètres sont envoyés : un identifiant, une référence au buffer et le nombre d'opérations à effectuer (production et/ou consommation) par l'objet
- lancez le programme avec un buffer initialement à 100 (avec limite MAX à 500) et 5 ProdConsos : que se passe-t-il ?
- Tout se passe bien ? Alors limiter le nombre de ProdConsos à 3 et lancez le programme avec un buffer initialement à 0.

```java
public class ProdConso extends Thread {

    int id, nbprodcons;
    Random alea;
    Buffer buffer;

    public ProdConso(int id, Buffer buffer, int nbprodcons) {
        this.id = id; this.nbprodcons = nbprodcons;
        this.buffer = buffer;
        alea = new Random(System.currentTimeMillis());
    }

    public void run() {
        int currentnb = 0;
        while(currentnb < nbprodcons) {
            if(alea.nextBoolean()) { // producteur
                System.out.println("prodcons:"+id+" va produire");
                buffer.ajouter();
                System.out.println("prodcons:"+id+" a fini de produire: nbprodcons="+currentnb);
                currentnb++;
            } else { // consommateur
                System.out.println("prodcons:"+id+" va consommer");
                buffer.retirer();
                System.out.println("prodcons:"+id+" a fini de consommer: nbprodcons="+currentnb);
                currentnb++;
            }
        }
        System.out.println("prodcons:"+id+" a TERMINE");
    }
}
```

```java
public class Lanceur {

    Buffer buf;
    ProdConso[] prodconsos;

    public static void main(String[] args) {
        new Lanceur(3);
    }

    public Lanceur(int nbProdConso) {
        buf = new Buffer(10);
        prodconsos = new ProdConso[nbProdConso];
        for(int p=0;p<nbProdConso;p++) {
            ProdConso pc = new ProdConso(p,buf,5);
            prodconsos[p] = pc;
            pc.start();
        }
    }
}
```

Observation

- Testez le programme plusieurs fois.
- Que constatez-vous ?
- Comment l'expliquez-vous ?

Trace d'exécution

- 3 instances de ProdConso buffer initialement à 10

```
prodcons :1 va produire
prodcons :2 va produire
prodcons :0 va produire
------- contenu buffer 11
prodcons :1 a fini de produire : nbprodcons=0
------- contenu buffer 12
prodcons :0 a fini de produire : nbprodcons=0
prodcons :0 va consommer
prodcons :1 va consommer
------- contenu buffer 13
prodcons :2 a fini de produire : nbprodcons=0
prodcons :2 va consommer
------- contenu buffer 12
prodcons :1 a fini de consommer : nbprodcons=1
prodcons :1 a TERMINE
------- contenu buffer 11
prodcons :0 a fini de consommer : nbprodcons=1
prodcons :0 a TERMINE
------- contenu buffer 10
prodcons :2 a fini de consommer : nbprodcons=1
prodcons :2 a TERMINE
```

Trace d'exécution

- 5 instances de ProdConso buffer initialement à 0

```
prodcons :0 va consommer
prodcons :3 va consommer
prodcons :2 va consommer
prodcons :1 va consommer
prodcons :4 va consommer
```

→ blocage