La programmation parallèle

La programmation concurrente en Java (2ème partie: exclusion mutuelle)

La programmation concurrente en Java (2ème partie)

- **Synchronisation**
  - `synchronized`
    - verrou sur une référence
    - Moniteur
  - barrière de synchronisation avec `wait`, `notify`, `notifyAll`
    - Producteurs/Consommateurs
    - Sémaphore

Multi-threading en Java

Problèmes:

- Avoir plusieurs activités posent des problèmes concernant:
  - l'ordonnancement : dans quel ordre les diverses activités sont exécutées ?
  - la synchronisation : comment les diverses activités peuvent se synchroniser ?
  - le partage : comment accéder aux données partagées par les diverses activités.

Synchronisation (1)

- **Section critique** : lorsque 2 threads ou plus ont besoin d'une même ressource au même moment, il y a besoin de s'assurer que la ressource ne sera utilisée que par un thread à un instant donné.

- **Exemple de problème** : si `i=2`, le code `i=i+1;`

  Exécuté par 2 threads, peut donner en fin d'exécution **3** ou **4** suivant l'ordre d'exécution : T1 lit la valeur de **i** (**2**),T2 lit la valeur de **i** (**2**)

  T1 calcule **i+1** (**3**) ,T2 calcule **i+1** (**3**)

  ==> le résultat final peut être **3** ou **4** !

- **Exemple d'exécution**:

```
Je suis dans le thread: main
Je suis bien dans le thread:T2
Je suis bien dans le thread:T1
T2: i = 3
main: i = 2
T1: i = 4
```

Synchronisation (2)

1/ Par héritage :

```
class Test2Threads {
 static int i=2;
 public static void main(String[] args) {
  Thread t1 = new Thread("T1") {
    public void run() {
      System.out.println("Je suis dans le thread: "+
        Thread.currentThread().getName());
      i = i + 1;
      System.out.println("T1: i = "+i);
    }
 };
 Thread t2 = new Thread("T2") {
   public void run() {
     System.out.println("Je suis dans le thread: "+
        Thread.currentThread().getName());
      i = i + 1;
      System.out.println("T2: i = "+i);
  }
 };
 t1.start(); t2.start();
/*
   try {
        t1.join();
        t2.join();
   } catch (InterruptedException e) {}
*/
// donne je suis dans le thread: main
     System.out.println("Je suis dans le thread: "+
       Thread.currentThread().getName());
     System.out.println("main: i = "+i);
 }
}
```

Résultat :

```
Je suis dans le thread: T2 Je
suis dans le thread: main Je
suis dans le thread: T1 main:
i = 2
T2: i = 3
T1: i = 4
Avec join(), main: i = 4
```

Synchronisation (3)

2/ par implémentation de Runnable :

```
public class Test2Runnables {
 static int i=2;
  public static void main(String[] args) {
   System.out.println("iInitiale= "+i);
   Runnable traitement = new Runnable() {
     public void run() {
        System.out.println("Je suis dans le thread:"+
             Thread.currentThread().getName());
        i=i+1;
        System.out.println(Thread.currentThread().
             getName()+": i="+i);
     }
   };

   Thread ta1 = new Thread(traitement, "tache1");
   ta1.start();

   Thread ta2 = new Thread(traitement, "tache2");
   ta2.start();
```

```
/*
  try {
        ta1.join();
        ta2.join();
  } catch (InterruptedException e) {}
*/
   System.out.println("Je suis dans le thread: "+
        Thread.currentThread().getName());
   System.out.println("iFinale= "+i);
 }
}
```

Résultat :

```
iInitiale= 2
Je suis dans le thread: main
iFinale= 2
Je suis dans le thread: tache1
tache1: i=3
Je suis dans le thread: tache2
tache2: i=4
```

Avec join(), iFinale = 4

Synchronisation (4)

- Pour s'assurer que la ressource ne sera utilisée que par un thread à un instant donné, on utilise un procédé de synchronisation avec le mot clé **`synchronized`**.

- Deux moyens permettent d'obtenir une synchronisation :
  - Méthode synchronisée **`m`** :

```
public synchronized type m(...) {
  // Le code de la méthode synchronisée.
  // aucun thread ne peut accéder à l'objet pour lequel la méthode est appelée
}
```

  - Bloc synchronisé sur l'objet **`o`** :

```
synchronized(o) {
  // Instructions de manipulation d'une ressource partagée.
  // aucun autre thread ne peut accéder à l'objet o
}
```

Synchronisation (5)

La sémantique de cette construction est de dire que le code ne sera exécuté que si le thread a le verrou du moniteur de l'objet .

Ainsi une méthode synchronisée :

```
public synchronized void methode0 {
      corps( ) ;
}
```

peut être vue comme une forme condensée de:

```
public void methode0 {
      synchronized(this) {
         corps( ) ;
      }
}
```

Méthode synchronisée (1)

- Exemple: Plusieurs threads veulent accéder à un ***compte en banque***

=> utiliser le mot clé **`synchronized`** qui fait en sorte que les méthodes soient exécutées en ***exclusion mutuelle***:

```
public class Compte
  static int solde = 0;
  public synchronized void deposer (int s) {
       int so = solde + s;
       solde = so;
  }
  public synchronized void retirer (int s) {
       int so = solde - s;
       solde = so;
  }
}
```

Méthode synchronisée (2)

2/ Sans synchronized :

```
public class Traitement1 {
   public static void main(String[] args) {
        Compte1 c = new Compte1();
        Thread t  = new Thread(new GestionCompte1(c));
        t.start();
   }
}
class GestionCompte1 implements Runnable {
   private Compte1 c;
   public GestionCompte1(Compte1 c){
     this.c = c;
   }
   public void run() {
    System.out.println("Solde initial:"+c.getSolde());
     for(int i =0; i <5; i++){
       if(c.getSolde() > 0){
          c.retraitArgent(20);
          System.out.println("Retrait effectué");
        }
     }
   }
}
```

```
class Compte1 {
   private int solde = 200;
   public int getSolde(){
       return this.solde;
   }
   public void retraitArgent(int retrait){
     solde = solde - retrait;
     System.out.println("Solde =" + solde);
   }
}
```

Résultat :

Solde initial: 200
Solde = 160
Retrait effectué
Solde = 140
Retrait effectué
Solde = 120
Retrait effectué …

Une simple boucle a pu faire le travail !

Méthode synchronisée (3)

2/ Sans synchronized :

```
public class Traitement2 {
   public static void main(String[] args) {
      Compte2 c1 = new Compte2();
      Compte2 c2 = new Compte2();
      Thread t1 = new Thread(new GestionCompte2(c1, "Mari"));
     Thread t2 = new Thread(new GestionCompte2(c2,"Epouse"));
     t1.start();
      t2.start();
   }
}
class  Compte2 {
   private int solde = 200;
   public int getSolde() {
      return this.solde;
   }
   public void retraitArgent(int retrait) {
      solde =solde - retrait;
      System.out.println("Solde =" + solde);
   }
}
```

Résultat :

```
Solde initial: 200 Solde initial: 200
Solde = 180 Retrait effectué par Mari
Solde = 160 Retrait effectué par Mari
```

```
class GestionCompte2 implements Runnable {
 private Compte2 c;
 private String name;
 public GestionCompte2(Compte2 c, String name){
   this.c = c;
   this.name = name;
 }
 public void run() {
  System.out.println("Solde initial:
          "+c.getSolde());
  for(int i =0; i <5; i++){
     if(c.getSolde() >0){ c.retraitArgent(20);
      System.out.println("Retrait effectué par"
         + this.name);
     }
  }
 }
}
```

Remarque:

~~Aucun problème~~ : nous avons utilisé deux instances distinctes de GestionCompte2 utilisant elles-mêmes deux instances distinctes de Compte2.

Méthode synchronisée (4)

2/ Sans synchronized :

```
public class Traitement2 {
   public static void main(String[] args) {
      Compte3 c = new Compte();
     Thread t1 = new Thread(new GestionCompte2(c, "Mari"));
     Thread t2 = new Thread(new GestionCompte2(c,"Epouse"));
      t1.start();
      t2.start();
   }
}
```

Résultat :

```
Solde initial: 200
Solde initial: 200
Solde  = 180
Solde  = 180
Retrait effectué par Epouse
Solde = 160
Retrait effectué par Epouse
Retrait effectué par Mari
Solde = 140
Retrait effectué par Epouse
Solde = 120
Retrait effectué par Epouse
Retrait effectué par Epouse
Solde = 160
```

Si nous avons utilisons deux instances distinctes de GestionCompte2 avec une seule instance de Compte2.

Remarque:

Une grande incohérence dans les résultats

Méthode synchronisée (5)

2/ Avec synchronized :

```
public synchronized void retraitArgent(int retrait) {
    solde = solde - retrait;
    System.out.println("Solde =" + solde);
}
```

Remarque:

Le modificateur `synchronized` permet un accès exclusif à la méthode `retraitArgent`.

Résultat :

```
Solde initial: 200 Solde initial: 200
Solde = 180
Solde = 160
Retrait effectué par Mari
Retrait effectué par Epouse
Solde = 140
Retrait effectué par Mari
Solde = 120
Retrait effectué par Epouse
Solde = 100
Retrait effectué par Epouse
Solde = 80
Retrait effectué par Mari
Solde = 60
Retrait effectué par Mari
Solde = 40
Retrait effectué par Mari
Solde = 20
Retrait effectué par Epouse
Solde = 0
Retrait effectué par Epouse
```

Bloc synchronisé (1)

Instruction `Synchronized`:

- Un appel synchronisé peut être réalisé dans un bloc synchronisé :

```
synchronized(objet) {
// instructions à synchroniser
}
```

- où `objet` est une référence à l'objet àsynchroniser.

- Un bloc synchronisé assure qu'un appel à une méthode quelconque d'`objet` ne pourra se faire qu'une fois que le thread courant sera entré dans le moniteur.

Bloc synchronisé (2)

- **Exemple 1**: Ecriture d'un programme mettant en jeu 3 threads qui utilisent une section critique avec un verrou d'un objet `v` d'une classe `Vide`.

Bloc synchronisé (3)

🗣 **Exemple 1:**

```
public class Synchroniser1 {
        public static void main(String[] args){
                Vide v = new Vide();
                Thread t1 = new Tache("T1", v);
                Thread t2 = new Tache("T2", v);
                Thread t3 = new Tache("T3", v);
                t1.start(); t2.start(); t3.start();
        }
}
class Vide { };
```

Résultat :

```
T3 entre dans la SC
T3 sort de la SC
T2 entre dans la SC
T2 sort de la SC
T1 entre dans la SC
T1 sort de la SC
```

```
class Tache extends Thread {
        Vide v;
        public Tache(String n, Vide v) {
                super(n);
                this.v = v;
        }
  public void run() {
     //   synchronized(v) {
        System.out.println(getName()+" entre dans la SC");
        System.out.println(getName()+" sort de la SC");
     //   }
   }
}
```

Remarque : Sans synchronized(v) plus d'un thread est dans la section critique

Bloc synchronisé (4)

- **Exemple 2:** Ecriture d'un programme mettant en jeu 2 threads dont l'un verrouille la sortie sur l'écran, l'objet (`System.out`).

```
public class Chiffres extends Thread {
  public void run() {
    try {
          synchronized(System.out) {
            for(int i=0; i<10; i++) {
                System.out.println(i);
                sleep(500);
            }
          }
    }
    catch(InterruptedException e) { }
  }
}
```

Bloc synchronisé (5)

```
public class Lettres extends Thread {
   public void run() {
        try {
                for(char c='a'; c<='f'; c++) {
                        System.out.println(c);
                        sleep(500);
                }
        }
        catch(InterruptedException e) { }
   }
}
public class Synchroniser2 {
   public static void main(String[] args) {
        Lettres A = new Lettres();
        Chiffres C = new Chiffres();
        A.start();
        C.start();
   }
}
```

Remarque : Dès que le thread a le moniteur de l'objet de type **Chiffre**, il verrouille la sortie.

Communication inter-threads `wait`, `notify`, `notifyAll (1)`

Dans un programme multitâches, les mécanismes de synchronisation précédents ne sont pas, en général, suffisants.

Exemples :

- Un thread **T**ne peut continuer son exécution que si une condition est remplie.
- Le fait que la condition soit remplie ou non dépend d'un autre thread.
  Par exemple, **T1** a besoin du résultat d'un calcul effectué par **T2**.

Dans de nombreux cas, il faut pouvoir mettre une ***activité en attente de conditions particulières*** et une activité réalisant cette condition devra alors réveiller les autres activités en attente.

Une solution coûteuse serait que **T1** teste la condition à intervalles réguliers.

Les méthodes **`wait()`** et **`notify()`** de la classe **`Object`** permettent de programmer efficacement ce genre de situation.

Communication inter-threads `wait`, `notify`, `notifyAll` (2)

Le mécanisme de communication inter-threads est accessible à travers 3 méthodes :

- La méthode `wait()` : fait sortir la tâche appelante du moniteur et la met en sommeil jusqu'à ce qu'un autre thread entre dans le même moniteur et appelle `notify()` ;

  Cette méthode bloque l'appelant.
- La méthode `notify()` : réveille le ***premier thread*** ayant appelé `wait()` sur le même moniteur;
- La méthode `notifyAll()` : réveille ***tous les threads*** ayant appelé `wait()` sur le même moniteur ; ***le thread de priorité la plus élevée s'exécutera en premier***.
- Ces méthodes sont implantées comme `final` dans `Object`, de sorte que toutes les classes en disposent.
- Ces trois méthodes doivent être appelées à l'intérieur d'une **méthode synchronisée** ou un **bloc synchronisé**.

Communication inter-threads `wait`, `notify`, `notifyAll` (3)

Déclaration : wait()

- `public final void wait()`
  `throws InterruptedException`

Utilisation :

- `objet.wait()`
  - Nécessite que le thread en cours possède le moniteur de **`objet`**.
  - Bloque le thread qui l'appelle, jusqu'à ce qu'un autre thread appelle la méthode **`objet.notify()`** ou **`objet.notifyAll()`**.
  - Libère le moniteur de l'objet.
  - Le code : `while(!condition) { objet.wait(); }`

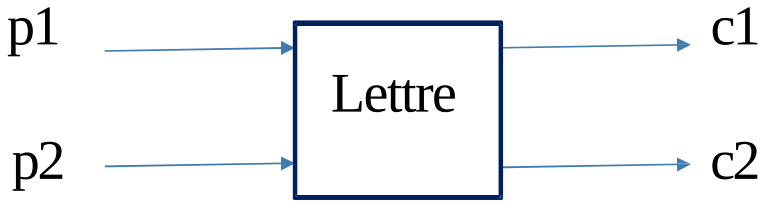
Remarque : L'opération du blocage du thread et libération du moniteur est atomique.

Il y a également la forme : `final void wait(long timeout)` qui permet d'attendre un nombre donné de millisecondes.

Exemple : Producteurs/Consommateurs (1)

- Un thread peut produire des résultats qui serviront de données à un autre thread.
- On peut utiliser un tampon intermédiaire
- Le thread producteur dépose ses résultats dans le tampon.
- Le thread consommateur prend ses données dans ce tampon.

- Nous allons considérer 2 classes de Threads, **`Producteur`** et **`Consommateur`** qui partagent une ressource commune: **`BoiteAuxLettres`**.
- Les producteurs vont déposer des valeurs dans une variable partagée (**`lettre`**) et les consommateurs vont les retirer.



Exemple : Producteurs/Consommateurs (2)

1- Exemple non synchronisé:

```
public class BoiteAuxLettres {
	private String lettre;

	public String retirer(String destinataire) {
		System.out.println("==> "+destinataire + "lit "+lettre);
		return lettre;
	}

	public void deposer(String lettre) {
		this.lettre = lettre;
		System.out.println("dépot de :"+lettre);
	}
}
```

Exemple : Producteurs/Consommateurs (3)

```
public class Producteur extends Thread {
        BoiteAuxLettres boite;
        String nom;
        public Producteur(BoiteAuxLettres boite, String nom) {
                this.boite = boite;
                this.nom = nom;
        }
        public void run() {
                for (int cpt = 1; cpt < 5; cpt++) {
                        try {
                                Thread.sleep((int) (Math.random()*2000));
                        }
                        catch(InterruptedException e) {
                        }
                        boite.deposer(nom+",lettre "+cpt);;
                }
        }
}
```

Exemple : Producteurs/Consommateurs (4)

```
public class Consommateur extends Thread {
	BoiteAuxLettres boite;
	String nom;

	public Consommateur(BoiteAuxLettres boite, String nom) {
		this.boite = boite;
		this.nom = nom;
	}
	public void run() {
		for(int cpt = 1; cpt < 5; cpt++) {
			try {
				Thread.sleep((int)(Math.random()*2000));
			}
			catch(InterruptedException e) {
			}
			boite.retirer(nom);
		}
	}
}
```

Exemple : Producteurs/Consommateurs (5)

```
public class Client {

	public static void main(String[] args) {
		BoiteAuxLettres b = new BoiteAuxLettres();
		Producteur p1 = new Producteur(b, "Sami");
		Producteur p2 = new Producteur(b, "Jalila");
		Consommateur c1 = new Consommateur(b, "Ali");
		Consommateur c2 = new Consommateur(b, "Raihana");
		p1.start();
		p2.start();
		c1.start();
		c2.start();
	}
}
```

Résultat : Les threads ne sont pas synchronisés,

- **Certaine données sont perdues (non consommées),**
- **Plusieurs consommations de la même donnée peuvent survenir avant une nouvelle production.**

Exemple bis: Producteurs/Consommateurs (6)

Exemple 1bis avec synchronisation :

```
public class BoiteAuxLettres_Syn {
        private boolean ok = false; // la boite aux lettres est vide
        private String lettre;

        public synchronized String retirer(String destinataire) {
                try {
                        while(!ok) wait(); // la boite aux lettres est vide
                }
                catch(InterruptedException e) {
                }
                System.out.println(destinataire + " lit "+lettre);
                ok = false; // la boîte aux lettres est de nouveau vidée
                notifyAll();
                return lettre;
        }
```

Exemple bis: Producteurs/Consommateurs (7)

Exemple 1bis avec synchronisation (suite) :

```
        public synchronized void deposer(String lettre) {
                try {
                        while(ok) wait();
                } // la boite est pleine, donc on ne peut pas déposer
                catch(InterruptedException e) {
                }
                this.lettre = lettre;
                System.out.println("dépot de:"+lettre);
                ok = true; // une lettre est de nouveau déposée
                notifyAll();
        }
}
```

Les sémaphores (1)

Un ***sémaphore*** est un mécanisme classique de verrouillage de ressource permettant d'assurer une forme d'***exclusion mutuelle***.

Plus précisément, un ***sémaphore*** est une valeur entière `s` associée à une file d'attente.

On peut accéder au ***sémaphore*** par deux méthodes:

- la première `P` attend que `s` soit *positif* et *décrémente* `s`, le thread accède à la section critique (*le thread est mis dans la file d'attente* si `s` **n'est pas strictement positif**),
- la deuxième `V` *incrémente* `s` ou *libère un des threads* attendant sur un `P`.

L'implémentation en Java suit directement cette définition:

- la méthode `P`, fera un `wait` pour attendre que `val` soit positive,
- alors que la méthode `V` incrémentera `val` et réveillera les threads en sommeil sur le `wait` par un `notifyAll`.

Les sémaphores (2)

Exemple 1: Le programme suivant réalise une ***exclusion mutuelle*** très simple utilisant un ***sémaphore***.

```
class Sem{
    int val;
    public Sem(int i){
        val=i;
    }
    synchronized void P(){
       try {
            while(val<=0){
                wait();
            }
            val--; // prend la main
       } catch(InterruptedException e){}
    }
    synchronized void V(){
       if(++val > 0) notifyAll(); // rend la main et réveille les autres
    }
}
```

Les sémaphores (3)

```
import java.util.Random;
class ThreadSem extends Thread{
     Sem mutex; // objet de type sémaphore
     public ThreadSem(String n, Sem s){
          super(n);
          mutex=s;
     }
     public void run(){
          int p;
          Random rand = new Random();
          while (true){
               try {
                    p=Math.abs(rand.nextInt()%5000);
                    sleep(p);
                    mutex.P();
                    System.out.println(getName()+ ": début de section critique" );
                    sleep(p);
                    System.out.println(getName()+ ": fin de section critique");
                    mutex.V();
               } catch (InterruptedException e) {}
          }
     }
}
```

Les sémaphores (4)

```
public class TestSemaphore{
    public static void main(String[] args) {
        Sem s= new Sem(1); // i=1 valeur positive, section critique accessible
        new ThreadSem("un",s).start();
        new ThreadSem("deux",s).start();
        new ThreadSem("trois",s).start();
     }
}
```

Résultat:

- deux: début de section critique
- deux: fin de section critique
- un: début de section critique
- un: fin de section critique

Le nombre de cœurs de la machine

Pour obtenir le nombre de coeurs d'une machine, Java propose une méthode dédiée dans : **`java.lang.runtime`**.

```
public class NombreDeCoeurs{
    public static void main(String[] args) {
        int nbDeCoeurs = Runtime.getRuntime().availableProcessors();
        System.out.println(nbDeCoeurs + " coeurs détéctés");
     }
}
```

Résultat:

- 4 coeurs détéctés