Calcul Parallèle

Cours 2 - JAVA et synchronisation

Eric Goubault
Commissariat à l'Energie Atomique & Chaire Ecole Polytechnique/Thalès

Le 12 janvier 2011

Plan du cours

- L'exclusion mutuelle
- `synchronized`, et les moniteurs/variables de condition: `wait()` et `notify()`
- Sémaphores
- Interblocage

Encore des comptes...

```
public class Compte {
    private int valeur;

    Compte(int val) {
        valeur = val;
    }

    public int solde() {
        return valeur;
    }
```



```
    public void depot(int somme) {
        if (somme > 0)
            valeur+=somme;
    }

    public boolean retirer(int somme)
           throws InterruptedException {
        if (somme > 0)
            if (somme <= valeur) {
                Thread.currentThread().sleep(50);
                valeur -= somme;
                Thread.currentThread().sleep(50);
                return true;
            }
        return false;
    } }
```

La banque...

```
public class Banque implements Runnable {
    Compte nom;

    Banque(Compte n) {
         nom = n; }

    public void Liquide (int montant)
           throws InterruptedException {
         if (nom.retirer(montant)) {
             Thread.currentThread().sleep(50);
```



```
             Donne(montant);
             Thread.currentThread().sleep(50); }
         ImprimeRecu();
         Thread.currentThread().sleep(50); }

    public void Donne(int montant) {
        System.out.println(Thread.currentThread().
getName()+": Voici vos " + montant + " euros."); }

    public void ImprimeRecu() {
        if (nom.solde() > 0)
            System.out.println(Thread.currentThread().
getName()+": Il vous reste " + nom.solde() + " euros.");
        else
            System.out.println(Thread.currentThread().
        getName()+": Vous etes fauches!");
```



```
    }

    public void run() {
        try {
           for (int i=1;i<10;i++) {
                Liquide(100*i);
                Thread.currentThread().sleep(100+10*i);
           }
        } catch (InterruptedException e) {
            return;
        }
    }

    public static void main(String[] args) {
        Compte Commun = new Compte(1000);
```



```
        Runnable Mari = new Banque(Commun);
        Runnable Femme = new Banque(Commun);
        Thread tMari = new Thread(Mari);
        Thread tFemme = new Thread(Femme);
        tMari.setName("Conseiller Mari");
        tFemme.setName("Conseiller Femme");
        tMari.start();
        tFemme.start();
    }
}
```

Une exécution

```
% java Banque
Conseiller Mari: Voici vos 100 euros.
Conseiller Femme: Voici vos 100 euros.
Conseiller Mari: Il vous reste 800 euros.
Conseiller Femme: Il vous reste 800 euros.
Conseiller Mari: Voici vos 200 euros.
Conseiller Femme: Voici vos 200 euros.
Conseiller Femme: Il vous reste 400 euros.
Conseiller Mari: Il vous reste 400 euros.
Conseiller Mari: Voici vos 300 euros.
Conseiller Femme: Voici vos 300 euros.
Conseiller Femme: Vous etes fauches!
Conseiller Mari: Vous etes fauches! ...
```


Résultat...

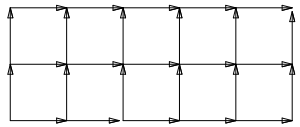
- Le mari a retiré 600 euros du compte commun,
- La femme a retiré 600 euros du compte commun,
- qui ne contenait que 1000 euros au départ!

Explication

("Sémantique")

- L'exécution de plusieurs threads se fait en exécutant une action insécable ("atomique") de l'un des threads, puis d'un autre ou d'éventuellement du même etc.
- Tous les "mélanges" possibles sont permis

Sémantique par entrelacements



Explication

Si les 2 threads `tMari` et `tFemme` sont exécutés de telle façon que dans `retirer`, chaque étape soit faite en même temps:

- Le test pourra trouvé être satisfait par les deux threads en même temps,
- qui donc retireront en même temps de l'argent.

Une solution

- Rendre "atomique" le fait de `retirer` de l'argent,
- Se fait en déclarant "synchronisée" la méthode `retirer` de la classe `Compte`:

```
public synchronized boolean retirer(int somme)
```


Maintenant...

```
% java Banque
Conseiller Mari: Voici vos 100 euros.
Conseiller Mari: Il vous reste 800 euros.
Conseiller Femme: Voici vos 100 euros.
Conseiller Femme: Il vous reste 800 euros.
Conseiller Mari: Voici vos 200 euros.
Conseiller Mari: Il vous reste 400 euros.
Conseiller Femme: Voici vos 200 euros.
Conseiller Femme: Il vous reste 400 euros.
Conseiller Femme: Il vous reste 100 euros.
Conseiller Mari: Voici vos 300 euros.
Conseiller Mari: Il vous reste 100 euros.
Conseiller Femme: Il vous reste 100 euros.
Conseiller Mari: Il vous reste 100 euros...
```


Résultat...

- Le mari a tiré 600 euros,
- La femme a tiré 300 euros,
- et il reste bien 100 euros dans le compte commun.

Remarque

- `synchronized` qualifie une méthode, mais est en fait un verrou au niveau de l'objet sur lequel s'applique la méthode
- dans le cas d'une méthode `static`, le verrou s'applique à la classe (i.e. toutes ses instances sont verrouillées)

Peut on se passer de synchronized?

```
public class CS1 extends Thread {
    Thread tour = null;

    public void AttendtonTour() {
        while (tour != Thread.currentThread()) {
            if (tour == null)
                tour = Thread.currentThread();
            try {
                Thread.sleep(100);
            } catch (Exception e) {}
        }
    }
```



```
public void RendlaMain() {
    if (tour == Thread.currentThread())
        tour = null;
}

public void run() {
    while(true) {
        AttendtonTour();
        System.out.println("C'est le tour de "+
               Thread.currentThread().getName());
        RendlaMain();
    } }
```



```
public static void main(String[] args) {
    Thread Un = new CS1();
    Thread Deux = new CS1();
    Un.setName("UN");
    Deux.setName("DEUX");
    Un.start();
    Deux.start();
} }
```

Problème: exécution synchrone des threads!

`wait()` et `notify()`

Chaque objet fournit un verrou, mais aussi un mécanisme de mise en attente (forme primitive de communication inter-threads; similaire aux variables de conditions ou aux moniteurs):

- `void wait()` attend l'arrivée d'une condition sur l'objet sur lequel il s'applique (en général `this` - mais pas seulement! voir après). Doit être appelé depuis l'intérieur d'une méthode ou d'un bloc `synchronized`, (il y a aussi une version avec timeout): on y revient aussi après!
- `void notify()` notifie un thread en attente d'une condition, de l'arrivée de celle-ci. De même, dans `synchronized`.
- `void notifyAll()` même chose mais pour tous les threads en attente sur l'objet.

Plus précisément

- Chaque objet JAVA *o* comporte une liste de threads en attente
- un thread y rentre en exécutant `o.wait()`
- un thread en sort si un `o.notify()` est exécuté par un autre thread, et que ce soit celui-ci dans la liste qui est libéré (en général, c'est le premier en attente qui est libéré)
- on doit absolument protéger l'accès à cette liste d'attente par un `synchronized`, le plus sûr étant un `synchronized(o)` (mais pas forcément, ce peut être sur la méthode faisant le `wait()` par exemple, si bien étudié...)

Un exemple (faux...)

Considérons :

```
class buffer1 {
 Object data = null;

 public synchronized void push(Object d) {
  try { if (data != null) wait();
   } catch (Exception e) { System.out.println(e); return;
  data = d;
  System.out.println("Pushed "+data);
  try { if (data != null) notify();
   } catch (Exception e) { System.out.println(e); return;
```


Un exemple...

```
public Object pop() {
 try { if (data == null) wait();
  } catch (Exception e) { System.out.println(e); return n
 Object o = data;
 System.out.println("Read "+o);
 data = null;
 try { if (data == null) notify();
 } catch (Exception e) { System.out.println(e); return nu
 return o; } }
```

Les threads producteur/consommateur

```
class Prod extends Thread {
 buffer1 buf;

 public Prod(buffer1 b) {
  buf = b; }

 public void run() {
  while (true) {
   buf.push(new Integer(1)); } } }
```


Les threads producteur/consommateur

```
class Cons extends Thread {
 buffer1 buf;

 public Cons(buffer1 b) {
  buf = b; }

 public void run() {
  while (true) {
   buf.pop(); } } }
```


Le main...

Construit 1 producteur et 2 consommateurs qui se partagent un buffer:

```
public class essaimon0 {
 public static void main(String[] args) {
  buffer1 b = new buffer1();
  new Prod(b).start();
  new Cons(b).start();
  new Cons(b).start(); } }
```


Première erreur...

```
is010046:Cours04new Eric$ javac essaimon0.java
is010046:Cours04new Eric$ java essaimon0 | more
Pushed 1
Read 1
java.lang.IllegalMonitorStateExceptionfrom pop-notify
java.lang.IllegalMonitorStateExceptionfrom pop-wait
java.lang.IllegalMonitorStateExceptionfrom pop-wait
...
```

Pas de `synchronized` dans la méthode `pop()`: non repéré à la compilation, mais à l'exécution!

Correction...

On rajoute `synchronized` dans la méthode `pop()`, puis:

```
is010046:Cours04new Eric$ javac essaimon0.java
is010046:Cours04new Eric$ java essaimon0 | more
Pushed 1
Read 1
Pushed 1
Read 1
Pushed 1
Read 1
Read null
Read null
...
```

Quel est le problème?

- `wait()` et `notify()` s'appliquent ici sur `this` qui est toujours le même et unique `buffer`
- Supposons que `data=null` (pas de `push` par exemple, ou un `pop` effectué)
- Quand le premier consommateur exécute `pop()`, il voit `data=null` et fait `wait()` : **il suspend sa prise de verrou sur le** `buffer`
- Le deuxième consommateur peut donc s'exécuter, voit `data=null` et fait `wait()` : **il suspend sa prise de verrou sur le** `buffer`
- Le producteur n'est pas bloqué, voit `data=null` et fait `data=1` puis `notify`, débloquant ainsi l'un des deux consommateurs, disons le premier
- Le premier consommateur lit alors `1` puis fait `data=null` et `notify()` et termine, libérant son verrou sur le `buffer`
- Le deuxième consommateur est alors débloqué et fait `read null`...

Une solution possible...

Utiliser deux objets, associé au buffer, qui signale "vide" et un autre qui signale "plein" (variables de conditions...):

```
Object full  = new Object();
Object empty = new Object();
Object data = null;
public void push(Object d) {
 synchronized(full) {
  try { if (data != null) full.wait();
   } catch (Exception e) { System.out.println(e); return
 data = d;
 System.out.println("Pushed "+data);
 synchronized(empty) {
  try { if (data != null) empty.notify();
   } catch (Exception e) { System.out.println(e); return
```


Solution...

```
public Object pop() {
 synchronized(empty) {
  try { if (data == null) empty.wait();
    } catch (Exception e) { System.out.println(e); return
 Object o = data;
 System.out.println("Read "+o);
 data = null;
 synchronized(full) {
  try { if (data == null) full.notify();
   } catch (Exception e) { System.out.println(e); return
 return o; } }
```

Exécution

```
is010046:Cours04new Eric$ javac essaimon1.java
is010046:Cours04new Eric$ java essaimon1 | more
Pushed 1
Read 1
Pushed 1
Read 1
Pushed 1
Read 1
...
```

(correct!)

Sémaphores

(en fait, cela existe déjà dans `java.util.concurrent.Semaphore`)

```
public class Semaphore {
    int n;
    String name;

    public Semaphore(int max, String S) {
        n = max;
        name = S;
    }

    public synchronized void P() {
        while (n == 0) {
            try {
                 wait();
            } catch(InterruptedException ex) {};
        }
        n--;
        System.out.println("P("+name+")");
    }

    public synchronized void V() {
        n++;
        System.out.println("V("+name+")");
        notify();
    }
}
```

Bien sûr ici, `this.wait` et `this.notify` sont ce que l'on souhaite... Remarquez le `while (n==0)` dans `void P()`.

Section Critique

```
public class essaiPV extends Thread {
    static int x = 3;
    Semaphore u;

    public essaiPV(Semaphore s) {
        u = s;
    }

    public void run() {
        int y;
        // u.P();
```

```
        try {
            Thread.currentThread().sleep(100);
            y = x;
            Thread.currentThread().sleep(100);
            y = y+1;
            Thread.currentThread().sleep(100);
            x = y;
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {};
        System.out.println(Thread.currentThread().
                           getName()+": x="+x);
        // u.V();
    }
```



```
    public static void main(String[] args) {
        Semaphore X = new Semaphore(1,"X");
        new essaiPV(X).start();
        new essaiPV(X).start();
    }
}
```


Résultat (sans P, V)

```
% java essaiPV
Thread-2: x=4
Thread-3: x=4
```


Résultat (avec P, V)

```
% java essaiPV
P(X)
Thread-2: x=4
V(X)
P(X)
Thread-3: x=5
V(X)
```

Remarque: ordonnancement tâches (cas 1 processeur)

- Le choix du thread JAVA à exécuter (partiellement): parmi les threads qui sont prêts.
- Ordonnanceur JAVA: ordonnanceur préemptif basé sur la priorité des processus.
- "Basé sur la priorité": essaie de rendre actif le(s) thread(s) prêt(s) de plus haute priorité.
- "Préemptif": interrompt le thread courant de priorité moindre, qui reste néanmoins prêt.

Ordonnancement de tâches

- Un thread actif qui devient bloqué, ou qui termine rend la main à un autre thread, actif, même s'il est de priorité moindre.
- La spécification de JAVA ne définit pas précisément la façon d'ordonnancer des threads de même priorité, par exemple:
  - "round-robin" (ou "tourniquet"): un compteur interne fait alterner l'un après l'autre (pendant des périodes de temps prédéfinies) les processus prêts de même priorité $\longrightarrow$ assure l'*équité*; aucune *famine* (plus tard...)
  - Ordonnancement plus classique (mais pas équitable) thread actif ne peut pas être préempté par un thread prêt de même priorité. Il faut que ce dernier passe en mode bloqué. (par `sleep()` ou plutôt `static void yield()`)

En fait...

L'ordonnancement dépend aussi bien de l'implémentation de la JVM que du système d'exploitation sous-jacent; 2 modèles principaux:

- "green thread", c'est la JVM qui implémente l'ordonnancement des threads qui lui sont associés (donc pas d'exploitation multi-processeur - souvent UNIX par défaut).
- "threads natifs", c'est le système d'exploitation hôte de la JVM qui effectue l'ordonnancement des threads JAVA (par défaut Windows).

Ressources partagées et priorités

Attention, il est possible de tomber dans le problème d'inversion de priorité:

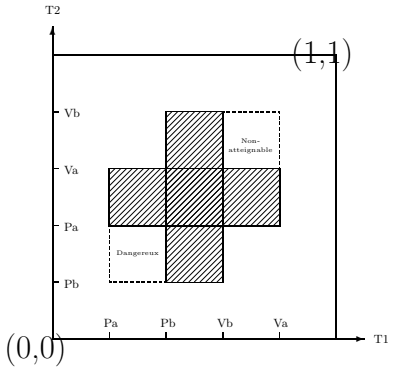
- Soit T_1 une tâche prioritaire par rapport à T_2, a et b deux objets partagés par T_1 et T_2 (protégés par exemple par deux sémaphores de même nom)
- Supposons l'exécution suivante:

T_1	T_2
Pa	-
(bloqué)	Pb
(obtient le verrou a)	(obtient le verrou b)
Pb	-
(bloqué)	-

- Ainsi, c'est T_2 qui s'exécute alors qu'il est de priorité moindre...

Un peu de sémantique

L'idée de base: "Progress graphs" (E.W.Dijkstra (1968))
`T1=Pa.Pb.Vb.Va` en parallèle avec `T2=Pb.Pa.Va.Vb`



"Image continue": x_i = temps local
Traces = chemins croissants dans chaque coordonnée = "di-chemins"

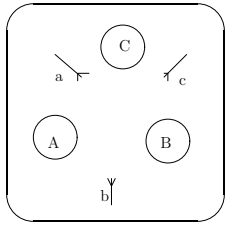
Portée de `synchronized`

- Purement syntaxique (contrairement à notre construction de `P` et `V`,
- On peut néanmoins raffiner et synchroniser un bloc d'instruction (sur un objet `obj`):

```
synchronized(obj) {
   ... }
```

- Une méthode `M` de la classe `A` qui est `synchronized` est équivalente à `P(A)`, corps de `M`, `V(A)`.

Philosophes qui dinent



Philosophes qui dinent

```
public class Phil extends Thread {
    Semaphore LeftFork;
    Semaphore RightFork;

    public Phil(Semaphore l, Semaphore r) {
        LeftFork = l;
        RightFork = r;
    }
```

```
    public void run() {
        try {
            Thread.currentThread().sleep(100);
            LeftFork.P();
            Thread.currentThread().sleep(100);
            RightFork.P();
            Thread.currentThread().sleep(100);
            LeftFork.V();
            Thread.currentThread().sleep(100);
            RightFork.V();
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {};
    }
}
```



```
public class Dining {

    public static void main(String[] args) {
        Semaphore a = new Semaphore(1,"a");
        Semaphore b = new Semaphore(1,"b");
        Semaphore c = new Semaphore(1,"c");
        Phil Phil1 = new Phil(a,b);
        Phil Phil2 = new Phil(b,c);
        Phil Phil3 = new Phil(c,a);
        Phil1.setName("Kant");
        Phil2.setName("Heidegger");
        Phil3.setName("Spinoza");
        Phil1.start();
        Phil2.start();
        Phil3.start();
    } }
```


Résultat

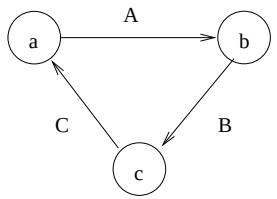
```
% java Dining
Kant: P(a)
Heidegger: P(b)
Spinoza: P(c)
^C
```


Interblocage

- Une exécution possible n'atteint jamais l'état terminal: quand les trois philosophes prennent en même temps leur fourchette gauche.
- Peut se résoudre avec un "ordonnanceur" extérieur (exercice classique).
- On pouvait s'en apercevoir sur le "request graph" ou sur le "progress graph": le point mort est du à l'intersection de 3 (=nombre de processus) hyperrectangles...

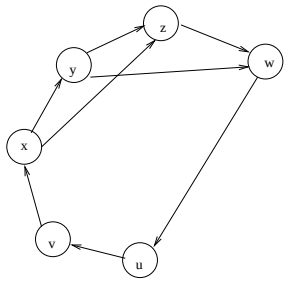
Request graph

Pa.Pb.Vb.Va|Pb.Pa.Va.Vb|Pc.Pa.Va.Vc:



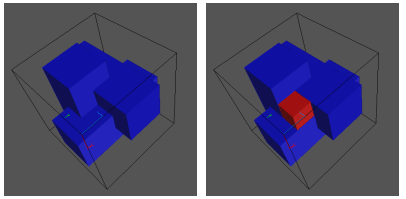
Autre exemple

```
P(x).P(y).P(z).V(x).P(w).V(z).V(y).V(w)|
P(u).P(v).P(x).V(u).P(z).V(v).V(x).V(z)|
P(y).P(w).V(y).P(u).V(w).P(v).V(u).V(v)
```



Explication

```
/* 3 philosophers ‘‘3p’’ */
A=Pa.Pb.Va.Vb
B=Pb.Pc.Vb.Vc
C=Pc.Pa.Vc.Va
```



Sémaphores d’ordre supérieur

```
public class essaiPVsup extends Thread {
    static int x = 3;
    Semaphore u;

    public essaiPVsup(Semaphore s) {
        u = s;
    }
```

```
    public void run() {
        int y;
        u.P();
        try {
            Thread.currentThread().sleep(100);
            y = x;
            Thread.currentThread().sleep(100);
            y = y+1;
            Thread.currentThread().sleep(100);
            x = y;
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {};
        System.out.println(Thread.currentThread().
                           getName()+": x="+x);
        u.V(); }
```



```
    public static void main(String[] args) {
        Semaphore X = new Semaphore(2,"X");
        new essaiPVsup(X).start();
        new essaiPVsup(X).start();
    }
}
```


Résultat

```
% java essaiPVsup
Thread-2: P(X)
Thread-3: P(X)
Thread-2: x=4
Thread-2: V(X)
Thread-3: x=4
Thread-3: V(X)
```

Pas de protection!

Réel intérêt

Pour les tampons de capacité bornée (problème du type producteur consommateur):

```
public class essaiPVsup2 extends Thread {
    static String[] x = {null,null};
    Semaphore u;

    public essaiPVsup2(Semaphore s) {
        u = s;
    }
```

```
public void push(String s) {
    x[1] = x[0];
    x[0] = s;
}

public String pop() {
    String s = x[0];
    x[0] = x[1];
    x[1] = null;
    return s; }

public void produce() {
    push(Thread.currentThread().getName());
    System.out.println(Thread.currentThread().
                       getName()+": push");
}
```



```
public void consume() {
    pop();
    System.out.println(Thread.currentThread().
                       getName()+": pop"); }

public void run() {
    try {
        u.P();
        produce();
        Thread.currentThread().sleep(100);
        consume();
        Thread.currentThread().sleep(100);
        u.V();
    } catch(InterruptedException e) {};
}
```



```
    public static void main(String[] args) {
        Semaphore X = new Semaphore(2,"X");
        new essaiPVsup2(X).start();
        new essaiPVsup2(X).start();
        new essaiPVsup2(X).start();
    }
}
```


Résultat

```
% java essaiPVsup2
Thread-2: P(X)
Thread-2: push
Thread-3: P(X)
Thread-3: push
Thread-2: pop
Thread-3: pop
Thread-2: V(X)
Thread-3: V(X)
Thread-4: P(X)
Thread-4: push
Thread-4: pop
Thread-4: V(X)
```