Parallélisme

Eric Goubault
Commissariat à l'Energie Atomique
Saclay

Plan du cours

- L'exclusion mutuelle
- `synchronized`, `wait()` et `notify()`
- Sémaphores
- Interblocage
- Contrôle avancé de threads, ordonnancement, machine JVM

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...

```java
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);
```



```java
            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 + " francs."); }

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



```java
    }

    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);
```



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


Résultat...

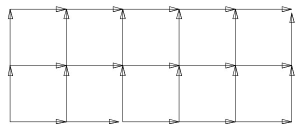
- Le mari a retiré 600 francs du compte commun,
- La femme a retiré 600 francs du compte commun,
- qui ne contenait que 1000 francs au départ!

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 francs.
Conseiller Mari: Il vous reste 800 francs.
Conseiller Femme: Voici vos 100 francs.
Conseiller Femme: Il vous reste 800 francs.
Conseiller Mari: Voici vos 200 francs.
Conseiller Mari: Il vous reste 400 francs.
Conseiller Femme: Voici vos 200 francs.
Conseiller Femme: Il vous reste 400 francs.
Conseiller Femme: Il vous reste 100 francs.
Conseiller Mari: Voici vos 300 francs.
Conseiller Mari: Il vous reste 100 francs.
Conseiller Femme: Il vous reste 100 francs.
Conseiller Mari: Il vous reste 100 francs...
```


Résultat...

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

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`). Doit être appelé depuis l'intérieur d'une méthode ou d'un bloc `synchronized`, (il y a aussi une version avec timeout)
- `void notify()` notifie un thread en attente d'une condition, de l'arrivée de celle-ci. De même, dans `synchronized`.
- `void notify_all()` même chose mais pour tous les threads en attente sur l'objet.

Sémaphores

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

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



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

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


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
```

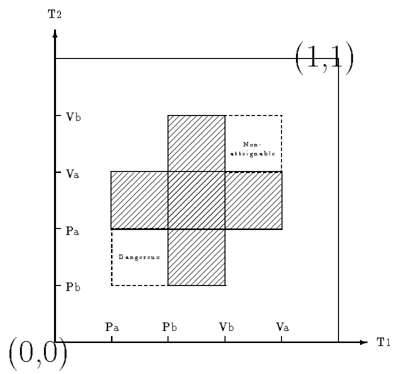

Résultat (avec P, V)

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


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)`.

POUR ÊTRE PLUS PRÉCIS...

- Toutes les variables (avec leurs verrous correspondants et la liste des threads en attente d'accès) de tous les threads sont stockées dans la mémoire gérée par la JVM correspondante.
- Les variables locales d'un thread ne peuvent pas être accédée par d'autres threads, donc on peut imaginer que chaque thread a une mémoire locale et a accès à une mémoire globale.
- Pour efficacité, chaque thread a copie locale des variables à utiliser $\longrightarrow$ mises à jour de la mémoire principale.

POUR ÊTRE PLUS PRÉCIS

- Plusieurs actions *atomiques* peuvent être effectuées pour gérer ces copies locales et la mémoire principale, au moins pour les types simples (`int`, `float`...).
  - *use* transfère le contenu de la copie locale d'une variable à la machine exécutant le thread.
  - *assign* transfère une valeur de la machine exécutant le thread à la copie locale d'une variable de ce même thread.

POUR ÊTRE PLUS PRÉCIS

- *load* affecte une valeur venant de la mémoire principale (après un *read*) à la copie locale à un thread d'une variable.
- *store* transfère le contenu de la copie locale à un thread d'une variable à la mémoire principale (qui va ainsi pouvoir faire un *write*).
- *lock* réclame un verrou associé à une variable à la mémoire principale.
- *unlock* libère un verrou associé à une variable.

EN FAIT...

Actions de la mémoire principale (la JVM):

- *read* transfère le contenu d'une variable de la mémoire principale vers la mémoire locale d'un thread (qui pourra ensuite faire un `load`).
- *write* affecte une valeur transmise par la mémoire locale d'un thread (par *store*) d'une variable dans la mémoire principale.

Pour les types "plus longs" comme `double` et `long`, la spécification permet tout à fait que *load*, *store*, *read* et *write* soient non-atomiques.

CONTRAINTES

Pour un thread donné:

- à tout *lock* correspond plus tard une opération *unlock*.
- chaque opération *load* est associée à une opération *read* qui la précède,
- chaque *store* est associé à une opération *write* qui la suit.

CONTRAINTES

Pour une variable donnée et un thread donné,

- l'ordre dans lequel des *read* sont faits par la mémoire locale (respectivement les *write*) est le même que l'ordre dans lequel le thread fait les *load* (respectivement les *store*).
- Vrai que pour une variable donnée!
- Pour les variables *volatile*, aussi respecté entre les *read/load* et les *write/store* entre les variables. Aussi: *use* suivent immédiatement les *load* et `store` suivent immédiatement les *assign*.

EXEMPLES

```
class Essai1 extends Thread {
  public Essai1(Exemple1 e) {
    E = e;  }

  public void run() {
    E.F1();  }

  private Exemple1 E;  }
```

Exemples

```
class Essai2 extends Thread {
  public Essai2(Exemple1 e) {
    E = e;  }

  public void run() {
    E.G1();  }

  private Exemple1 E;  }
```


Exemples

```
class Lance {
  public static void main(String args[]) {
    Exemple1 e = new Exemple1();

    Essai1 T1 = new Essai1(e);
    Essai2 T2 = new Essai2(e);

    T1.start();
    T2.start();
    System.out.println("From Lance a="+e.a+" b="+e.b);  } }
```


Exemples

```
class Exemple1 {
  int a = 1;
  int b = 2;

  void F1() {
    a = b;  }

  void G1() {
    b = a;  }  }
```


Exemples

```
class Exemple2 {
  int a = 1;
  int b = 2;

  synchronized void F2() {
    a = b;  }

  synchronized void G2() {
    b = a;  } }
```

EXÉCUTIONS

```
> java Lance
a=2 b=2
> java Lance
a=1 b=1
> java Lance
a=2 b=1
```


EXPLICATION

Les chemins possibles d'exécution sont les entrelacements des 2 traces séquentielles:

$$read\, b \longrightarrow load\, b \longrightarrow use\, b \longrightarrow assign\, a \longrightarrow store\, a \longrightarrow write\, a$$

pour le thread `Essai1` et

$$read\, a \longrightarrow load\, a \longrightarrow use\, a \longrightarrow assign\, b \longrightarrow store\, b \longrightarrow write\, b$$

pour le thread `Essai2`.

EXPLICATION

Appelons $a1, b1$ (respectivement $a2$ et $b2$) les copies locales des variables a et b pour le thread `Essai1` (respectivement pour le thread `Essai2`):

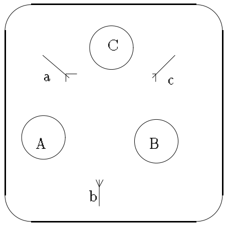
$$\begin{array}{c} write\, a \longrightarrow read\, a \longrightarrow read\, b \longrightarrow write\, b \\ read\, a \longrightarrow write\, a \longrightarrow write\, b \longrightarrow read\, b \\ read\, a \longrightarrow write\, a \longrightarrow read\, b \longrightarrow write\, b \end{array}$$

EXPLICATION

- l'état de la machine est $la1 = 2$, $lb1 = 2$, $a = 2$, $b = 2$, $la2 = 2$ et $lb2 = 2$,
- ou $la1 = 1$, $lb1 = 1$, $a = 1$, $b = 1$, $la2 = 1$ et $lb2 = 1$.
- ou $la1 = 2$, $lb1 = 2$, $a = 2$, $b = 1$, $la1 = 1$ et $lb1 = 1$.

Si on utilise la version synchronisée de `Exemple1`: `Exemple2`, alors on n'a plus que les deux premiers ordonnancements qui soient possibles.

Philosophes qui dînent



Philosophes qui dînent

```
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

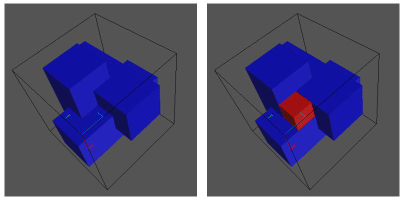
```
% java Dining
Kant: P(a)
Heidegger: P(b)
Spinoza: P(c)
^C
```


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...

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)
```

Complément: Priorités et Ordonnancement

- `void setPriority(int priority)` assigne une priorité au thread donné
- `int getPriority()` renvoie la priorité d'un thread donné
- `static void yield()`: le thread courant rend la main, ce qui permet à la machine virtuelle JAVA de rendre actif un autre thread de même priorité

Etats d'un thread

Un thread peut être dans l'un des 4 cas suivants:

- l'état initial, entre sa création et `start()`.
- l'état prêt, immédiatement après `start`.
- l'état bloqué: thread en attente, d'un verrou, d'un socket, quand il est suspendu (`sleep(long)`
- l'état terminaison, quand `run()` se termine ou quand on invoque `stop()` (à éviter si possible... n'existe plus en JAVA 2).

Etats d'un thread

- Détermination de l'état: `isAlive()`, qui renvoie un booléen indiquant si un thread est encore vivant, c'est à dire s'il est prêt ou bloqué
- On peut aussi interrompre l'exécution d'un thread qui est prêt (passant ainsi dans l'état bloqué). `void interrupt()` envoie une interruption au thread spécifié; si pendant `sleep`, `wait` ou `join`, lèvent une exception `InterruptedException`.

Etats d'un thread

- `void join()` attend terminaison d'un thread. Egalement: `void join(long timeout)` attend au maximum `timeout` millisecondes.

```
public class ... extends Thread {
   ...
   public void stop() {
      t.shouldRun=false;
      try {
         t.join();
      } catch (InterruptedException e) {}   } }
```

Priorités

- Priorité = nombre entier, qui plus il est grand, plus le processus est prioritaire.
- `void setPriority(int priority)` assigne une priorité et `int getPriority()` renvoie la priorité.
- La priorité peut être maximale: `Thread.MAX_PRIORITY`, normale (par défaut): `Thread.NORM_PRIORITY` (au minimum elle vaut 0).

Processus démons

On peut également déclarer un processus comme étant un démon ou pas:

```
setDaemon(Boolean on);
boolean isDaemon();
```

"support" aux autres (horloge, ramasse-miettes...). Il est détruit quand il n'y a plus aucun processus utilisateur (non-démon) restant

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'ordonna 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 ⟶ 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).