

## Parallélisme

ERIC GOUBAULT  
COMMISSARIAT À L'ÉNERGIE ATOMIQUE & CHAIRE ECOLE  
POLYTECHNIQUE/THALÈS

---

## 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...

```

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 + " 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!");
    }
}

```

```

}

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 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! ...
```

9

## 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!

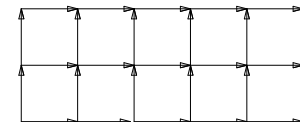
## EXPLICATION

(“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

11

## 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.

13

## 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...
```

15

## 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) {}
        }
    }
}
```

17

```
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!

19

### 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;
    }
}
```

21

```
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();
    }
}
```

23

```
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();
}
}

```

25

RÉSULTAT (AVEC P, V)

```

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

```

27

RÉSULTAT (SANS P, V)

```

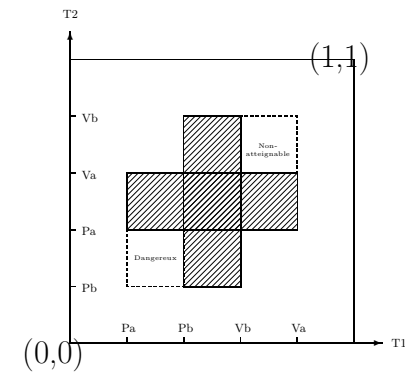
% java essaiPV
Thread-2: x=4
Thread-3: x=4

```

### 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  $M$ ,  $V(A)$ .

29

## 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 accédées 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 → 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.

31

## 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; }
```

37

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; } }
```

39

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
```

41

## EXPLICATION

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

$readb \longrightarrow loadb \longrightarrow useb \longrightarrow assigna \longrightarrow storea \longrightarrow writea$

pour le thread **Essai1** et

$reada \longrightarrow loada \longrightarrow usea \longrightarrow assignb \longrightarrow storeb \longrightarrow writeb$

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**):

$readb \longrightarrow writea \longrightarrow reada \longrightarrow writeb$   
 $reada \longrightarrow writeb \longrightarrow readb \longrightarrow writea$   
 $reada \longrightarrow readb \longrightarrow writeb \longrightarrow writea$

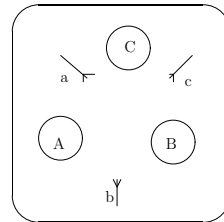
43

## 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 DINENT



45

## 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) {}
}
}
```

47

```
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
```

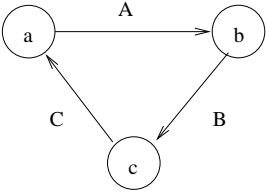
49

## 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

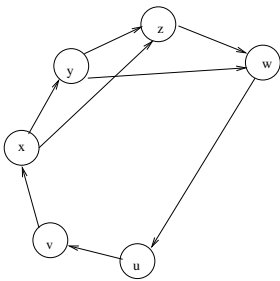
$Pa.Pb.Vb.Va | Pb.Pa.Va.Vb | Pc.Pa.Va.Vc:$



51

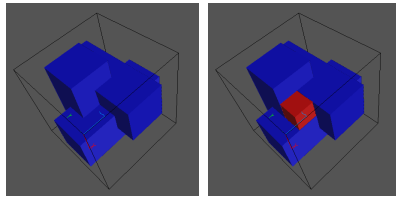
## AUTRE EXEMPLE

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



## EXPLICATION

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



53

## 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(); }  
}
```

55

```
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!
```

57

## 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");
}
```

59

```
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();
}
}
```

---

61

## 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)
```