Parallélisme

Eric Goubault
Commissariat à l'Energie Atomique
Saclay

Plan du cours

- Ordonnancement de tâches JAVA
- Le problème de l'exclusion mutuelle
- `synchronized`, `wait()` et `notify()`
- Sémaphores
- Sémantique, interblocage et séquentialisation
- Simulation de passage de messages
- Contrôle avancé de threads, application: barrières de synchronisation

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'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).

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! ...
```


Résultat...

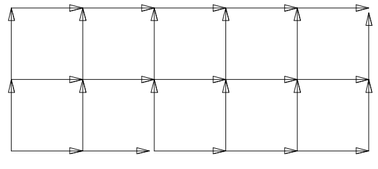
- Le mari a retiré 600 francs du compte commun,
- La femme a retiré 600 francs du compte commun,
- qui ne contenait que 1000 francs au départ!

Explication

("Sémantique" – certes idéalisée...)

- 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.

Sémantique

Langage simple, "fragment de JAVA":

EXPR	::=	CONSTANTE	VARIABLE
	|	EXPR+EXPR	EXPR*EXPR
	|	EXPR/EXPR	EXPR-EXPR
TEST	::=	EXPR==EXPR	EXPR < EXPR
	|	EXPR > EXPR	TEST && TEST
	|	TEST || TEST	! TEST
INSTR	::=	VARIABLE=EXPR	

Sémantique

BLOCK ::= ϵ
| INSTR;BLOCK
| if TEST
then BLOCK
else BLOCK ;BLOCK
| while TEST
BLOCK ;
BLOCK

Systèmes de transitions

Sémantique d'un `BLOCK` inductivement sur la syntaxe, en lui associant ce que l'on appelle un système de transitions: 4-uplet $(S, i, E, Tran)$ où,

- S est un ensemble d'états,
- $i \in S$ est l'"état initial",
- E est un ensemble d'étiquettes,
- $Tran \subseteq S \times E \times S$ est l'ensemble des transitions. On note plus communément $(s, t, s') \in Tran$ comme $s \xrightarrow{t} s'$.

Sémantique

- Environnement: toute fonction de `VARIABLE` vers l'ensemble des entiers relatifs.
- Un état du système de transition sera une paire (l, ρ) où $l \in L$ (ce qu'il reste à exécuter) et ρ est un environnement.

Sémantique

- $(x = expr; block, \rho) \xrightarrow{x = expr} (block, \rho[x \rightarrow [\![expr]\!](\rho)])$
  où $[\![expr]\!](\rho)$: résultat de l'évaluation de l'expression $expr$ dans l'environnement ρ
- $(if\ \ t\ \ then\ \ b_1\ \ else\ \ b_2; b_3, \rho) \xrightarrow{then} (b_1; b_3, \rho)$
  si $[\![t]\!]\rho = vrai$ sinon

$$(if\ \ t\ \ then\ \ b_1\ \ else\ \ b_2; b_3, \rho) \xrightarrow{else} (b_2; b_3, \rho)$$

Sémantique

- $(while\ \ t\ \ b_1;\ \ b_2, \rho) \xrightarrow{while} (b_1; while\ \ t\ \ b_1;\ \ b_2, \rho)$
  si $[\![t]\!]\rho = vrai$ sinon

$$(while\ \ t\ \ b_1;\ \ b_2, \rho) \xrightarrow{while} (b_2, \rho)$$

Entrelacements

Ensemble de processus $P_1, \ldots, P_n \in L$. "Entrelacements":

- $P_j \longrightarrow (S_j, i_j, E_j, Tran_j)$,
- $(S, i, E, Tran) = [\![P_1 \mid \ldots \mid P_n]\!]$ par:
  - $S = S_1 \times \ldots \times S_n$, $i = (i_1, \ldots, i_n)$,
  - $E = E_1 \cup \ldots \cup E_n$,
  - $Tran = \{ (s_1, \ldots, s_j, \ldots, s_n) \xrightarrow{t_j} (s_1, \ldots, s'_j, \ldots, s_n)$ où $s_j \xrightarrow{t_j} s'_j \in Tran_j \}$

Exécutions

- Les exécutions ou traces ou entrelacements dans cette sémantique sont des suites [finies]:

$$s_0 = i \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_n} s_n$$

  telles que $s_{i-1} \xrightarrow{t_i} s_i \in Tran$ pour tout i entre 1 et n.

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

On rajouter à L: $P(VARIABLE)$ et $V(VARIABLE)$. La sémantique par systèmes de transitions change assez peu:

- On rajoute dans état global S une composante nouvelle: fonction $\kappa : VARIABLE \to \{0, 1\}$.
- $i = (P_1, \ldots, P_n, \kappa_0)$ où κ_0: fonction uniforme $= 1$,
- E: nouvelles actions Px et Vx,

Sémantique

- nouvelles transitions:

$$(b_1, \ldots, Px; b_j, \ldots, b_n, \kappa) \xrightarrow{Px} (b_1, \ldots, b_j, \ldots, b_n, \kappa')$$

si $\kappa(x) = 1$; et alors $\kappa'(y) = \kappa(x)$ pour tout $y \neq x$ et $\kappa'(x) = 0$, et,

$$(b_1, \ldots, Px; b_j, \ldots, b_n, \kappa) \xrightarrow{Vx} (b_1, \ldots, b_j, \ldots, b_n, \kappa')$$

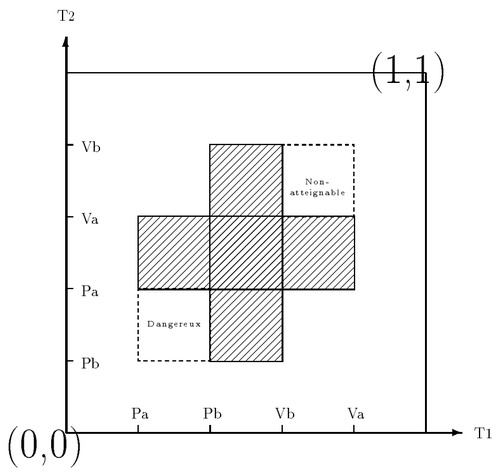
avec $\kappa'(y) = \kappa(x)$ pour tout $y \neq x$ et $\kappa'(x) = 1$.

L'apparition de ces nouvelles transitions contraint les entrelacements possibles.

Une "abstraction"

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`):

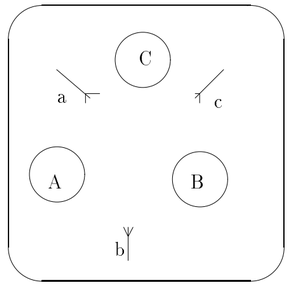
$$write\,a \longrightarrow read\,a \longrightarrow read\,b \longrightarrow write\,b$$
$$read\,a \longrightarrow write\,a \longrightarrow write\,b \longrightarrow read\,b$$
$$read\,a \longrightarrow write\,a \longrightarrow read\,b \longrightarrow write\,b$$

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

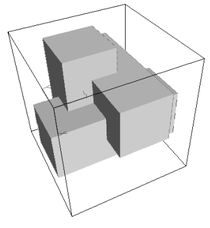
```
% java Dining
Kant: P(a)
Heidegger: P(b)
Spinoza: P(c)
^C
```

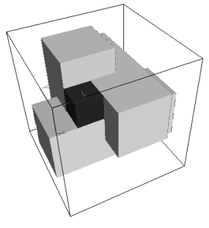

Interblocage

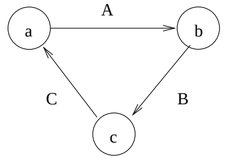
- Une exécution possible n'atteint jamais l'état terminal: quand les trois philosophes prennent en même temps leur fourchette gauche.
- Peut se résoudre avec un "ordonnanceur" extérieur (exercice classique).
- On pouvait s'en apercevoir sur le "request graph" ou sur le "progress graph": le point mort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





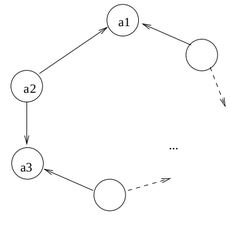


Une solution possible

- Numéroter les philosophes,
- Imposer que les philosophes pairs acquièrent la fourchette gauche d'abord, puis la droite et impairs acquièrent la fourchette droite d'abord, puis la gauche.

Une solution

```
P1=Pa2.Pa1.Va2.Va1
P2=Pa2.Pa3.Va2.Va3
...
P2k=Pa2k.Pa(2k+1).Va2k.Va(2k+1)
P(2k+1)=Pa(2k+2).Pa(2k+1).Va(2k+2).Va(2k+1)
```



Sémaphores à compteur

```
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)
```

```
public class Ticket {
    String nom;}
public class Requete extends Thread {
    Ticket Aller;
    Ticket Retour;
    Semaphore u;
    Semaphore v;
    public Requete(Ticket t, Ticket s,
                   Semaphore x, Semaphore y) {
        Aller = t;
        Retour = s;
        u = x;
        v = y; }
```



```
    public void run() {
        try {
            u.P();
            Thread.currentThread().sleep(100);
            if (Aller.nom == null)
                 Aller.nom = Thread.currentThread().
                                    getName();
            Thread.currentThread().sleep(100);
            u.V();
        } catch (InterruptedException e) {};
```



```
        try {
            v.P();
            Thread.currentThread().sleep(100);
            if (Retour.nom == null)
                Retour.nom = Thread.currentThread().
                                    getName();
            Thread.currentThread().sleep(100);
            v.V();
        } catch (InterruptedException e) {};
    }
}
```



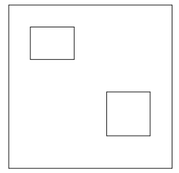
```
public class Serial {
    public static void main(String[] args) {
Semaphore a = new Semaphore(1,"a");
Semaphore b = new Semaphore(1,"b");
Ticket s = new Ticket();
Ticket t = new Ticket();
Requete Bill = new Requete(s,t,a,b);
Requete Tom = new Requete(t,s,b,a);
Bill.setName("Bill");
Tom.setName("Tom");
Bill.start();
Tom.start();
try {
    Tom.join();
    Bill.join();
} catch(InterruptedException e) {};
```

```
        System.out.println("Le billet Paris->New York
                           est au nom de: "+s.nom);
        System.out.println("Le billet New York->Paris
                           est au nom de: "+t.nom); } }
```


Résultat...

```
% java Serial
Bill: P(a)
Tom: P(b)
Bill: V(a)
Tom: V(b)
Bill: P(b)
Tom: P(a)
Bill: V(b)
Tom: V(a)
Le billet Paris->New York est au nom de: Bill
Le billet New York->Paris est au nom de: Tom
```


Progress graph



2-phase locking

Changer dans **Requete**, l'ordre des verrous, dans la méthode **run()**:

```
    public void run() {
        try {
            u.P();
            v.P();
            Thread.currentThread().sleep(100);
            if (Aller.nom == null)
                Aller.nom = Thread.currentThread().
                                   getName();
            Thread.currentThread().sleep(100);
```

```
            Thread.currentThread().sleep(100);
            if (Retour.nom == null)
                Retour.nom = Thread.currentThread().
                                        getName();
            Thread.currentThread().sleep(100);
            u.V();
            v.V();
        } catch (InterruptedException e) {};
    }
```


Résultat

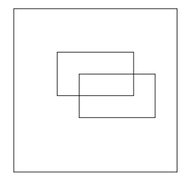
```
% java Serial2
Bill: P(a)
Bill: P(b)
Bill: V(a)
Bill: V(b)
Tom: P(b)
Tom: P(a)
Tom: V(b)
Tom: V(a)
Le billet Paris->New York est au nom de: Bill
Le billet New York->Paris est au nom de: Bill
```


Mais...

Il est possible qu'une exécution donne également un interblocage...

```
% java Serial2
Bill: P(a)
Tom: P(b)
^C
```


Progress graph



Les seules obstructions sont "au centre": on peut déformer tout chemin sur l'un des bords.

Exemple d'utilisation

L'interface `Serializable` de JAVA:

- permet d'assurer la persistence de certains objets,
- utilisé en particulier pour les entrées sorties

Simulation de passage par messages

```
public class MsgQueue {
    Vector queue = new Vector();

    public synchronized void send(Object obj) {
        queue.addElement(obj);  }

    public synchronized Object recv() {
        if (queue.size() == 0)
            return null;
        Object obj = queue.firstElement();
        queue.removeElementAt(0);
        return obj; } }
```


Exemple

```
public class Process {
    MsgQueue In;
    MsgQueue Out;

    public Process(MsgQueue i, MsgQueue o) {
        In = i;
        Out = o;
    }
```



```
    public void send(int x) {
        Out.send(new Integer(x));
        System.out.println(Thread.currentThread().
                           getName()+": send("+x+")"); }

    public int recv() {
        Object x = In.recv();
        while (x == null)
            x = In.recv();
        int res = ((Integer) x).intValue();
        System.out.println(Thread.currentThread().
                           getName()+": recv "+res);
        return res; }
}
```

Réception bloquante, "active polling".

Exemple

```
public class essaiMsg extends Thread{
    Process p;
    public essaiMsg(Process q) {
        p = q; }

    public void run() {
        for (int i=1;; i++) {
            p.send(i);
            p.recv(); } }
```



```
    public static void main(String[] args) {
        MsgQueue Canal1 = new MsgQueue();
        MsgQueue Canal2 = new MsgQueue();
        Process Tim = new Process(Canal1, Canal2);
        Process Tom = new Process(Canal2, Canal1);
        new essaiMsg(Tim).start();
        new essaiMsg(Tom).start();
    }
}
```


Résultat

```
% java essaiMsg
Thread-2: send(1)
Thread-3: send(1)
Thread-3: recv 1
Thread-3: send(2)
Thread-2: recv 1
Thread-2: send(2)
Thread-2: recv 2
Thread-2: send(3)
Thread-3: recv 2
^C
```


Barrières de synchronisation

(ref.: "Java Threads", O'Reilly)

```
    public Barrier(int nThreads) {
        threads2Wait4 = nThreads;
    }

    public synchronized int waitForRest()
        throws InterruptedException {
        int threadNum = --threads2Wait4;
```

```
    if (iex != null) throw iex;
    if (threads2Wait4 <= 0) {
        notifyAll();
        return threadNum;
    }
    while (threads2Wait4 > 0) {
        if (iex != null) throw iex;
        try {
            wait();
        } catch(InterruptedException ex) {
            iex = ex;
            notifyAll();
        }
    }
    return threadNum;
}
```



```
    public synchronized void freeAll() {
        iex = new InterruptedException("Barrier Released
                                        by freeAll");
        notifyAll();
    }
}
```


Exemple d'utilisation

```
public class ProcessIt implements Runnable {
    String[] is;
    Barrier bpStart, bp1, bp2, bpEnd;
    public ProcessIt(String[] sources) {
        is = sources;
        bpStart = new Barrier(sources.length);
        bp1 = new Barrier(sources.length);
        bp2 = new Barrier(sources.length);
        bpEnd = new Barrier(sources.length);
        for (int i=0;i<is.length; i++) {
            (new Thread(this)).start(); } }
```



```
public void run() {
    try {
        int i = bpStart.waitForRest();
        doPhaseOne(is[i]);
        bp1.waitForRest();
        doPhaseTwo(is[i]);
        bp2.waitForRest();
        doPhaseThree(is[i]);
        bpEnd.waitForRest();
    } catch(InterruptedException ex) {};
}

public void doPhaseOne(String ps) {
}
```

```
    public void doPhaseTwo(String ps) {
    }

    public void doPhaseThree(String ps) {
    }

    static public void main(String[] args) {
        ProcessIt pi = new ProcessIt(args);
    }
}
```