



Systèmes et Applications Répartis

M1 Informatique / M1 MIAGE
Université d'Evry Val d'Essonne

Tarek Melliti (tarek.melliti@ibisc.fr)

Pascal Poizat (pascal.poizat@lri.fr)

Bachir Djafri (bachir.djafri@univ-evry.fr)

Objectifs du cours

- Introduction et motivations
- Concurrence intra-application (processus, threads)
- Concurrence inter-applications
 - connecteurs de base (sockets)
 - appels de procédures/méthodes distantes (RPC, RMI)
 - objets distribués (RMI & CORBA)

important:

Le cours porte sur l'**analyse** et la programmation de systèmes et d'applications répartis

Le terme d'**analyse** (et par la-même, de conception) est aussi important que celui de la **programmation**

Rappel

- un système réparti est une collection de **processus indépendants et coopératifs** s'exécutant sur une ou plusieurs machines et apparaissant comme un seul et unique système cohérent
- cas 1 : concurrence intra (ou parallélisme)
 - plusieurs processus sur une même machine/app
- cas 2 : concurrence inter (ou répartition)
 - plusieurs processus sur plusieurs machines
 - pas d'horloge ni de mémoire commune

Pourquoi le parallélisme (dans ce cours)

- forme simple de système réparti, permettant d'aborder : concurrence - synchronisation - états cohérents
- permet de simuler la répartition par le biais de «pipes»
- permet de comprendre l'importance de l'analyse des systèmes et des applications répartis
- illustration avec les processus léger en Java (threads)

Processus lourds vs légers

- un processus lourd à son propre environnement d'exécution (espace mémoire)
- un processus léger partage son environnement d'exécution avec d'autres processus (processeur, mémoire, d'autres ressources)

Processus lourds vs légers

- efficacité
 - création des processus légers plus rapidement
 - plusieurs processus pour différentes tâches, mais communication entre processus légers plus simple
- espace mémoire
 - un processus lourd a son propre espace mémoire
 - les processus légers partagent l'espace mémoire

Processus léger en Java

- un processus léger (thread) est un objet (nous sommes en Java !)
- une application Java peut contenir plusieurs threads
- il existe toujours un premier thread (main / JVM)
- deux possibilités :
 - instance de la classe Thread ou d'une sous classe définie par le programmeur - problème : absence d'héritage multiple en Java
 - instance d'une classe qui implante l'interface Runnable (un thread est alors attaché à l'objet)

Processus légers en Java

- avec Thread :

```
class MonThread extends Thread { ... }  
MonThread p1 = new MonThread( ... );
```

- avec Runnable :

```
class MonThread extends ... implements Runnable { ... }  
MonThread p1 = new MonThread( ....);
```

- il faut définir la méthode `run()`, c'est le «code» du thread
- le lancement du thread se fait avec la méthode `start()`

Exemple : balle v1 (1/2)

```
class Ball {  
    private Component canvas;  
    private static final int XSIZE = 15;  
    private static final int YSIZE = 15;  
    private int x = 0;  
    private int y = 0;  
    private int dx = 2;  
    private int dy = 2;  
    public static boolean freeze = false;  
  
    public Ball(Component c) { canvas = c; }  
  
    public void draw(Graphics2D g2) {  
        g2.fill(new Ellipse2D.Double(x, y, XSIZ  
    }  
  
    public void go() {  
        try {  
            while (true) {  
                if (!freeze)  
                    move();  
            }  
        }  
        catch (InterruptedException exception) {}  
    }  
  
    public void move() {  
        x += dx;  
        y += dy;  
        if (x < 0) {  
            x = 0;  
            dx = -dx;  
        }  
        if (x + XSIZ  
        x = canvas.getWidth() - XSIZ  
        dx = -dx;  
    }  
    if (y < 0) {  
        y = 0;  
        dy = -dy;  
    }  
    if (y + YSIZE >= canvas.getHeight()) {  
        y = canvas.getHeight() - YSIZE;  
        dy = -dy;  
    }  
    canvas.paint(canvas.getGraphics());  
}
```

Exemple : balle v1 (2/2)

```
class BounceFrame extends JFrame {
    private BallCanvas canvas;

    public BounceFrame() {
        setSize(450, 350);
        setTitle("Bounce");
        Container contentPane = getContentPane();
        canvas = new BallCanvas();
        contentPane.add(canvas, BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        addButton(buttonPanel, "New",
            new ActionListener() {
                public void actionPerformed(ActionEvent evt){
                    canvas.addBall();
                }
            });
        addButton(buttonPanel, "Start/Stop",
            new ActionListener() {
                public void actionPerformed(ActionEvent evt){
                    Ball.freeze = !Ball.freeze;
                }
            });
        contentPane.add(buttonPanel, BorderLayout.SOUTH);
    }

    public void addButton(Container c, String title,
        ActionListener listener) {
        JButton button = new JButton(title);
        c.add(button);
        button.addActionListener(listener);
    }
}

class BallCanvas extends JPanel {
    private ArrayList balls = new ArrayList();

    public void addBall() {
        Ball b = new Ball(this);
        balls.add(b);
        b.go();
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        for (int i = 0; i < balls.size(); i++) {
            Ball b = (Ball)balls.get(i);
            b.draw(g2);
        }
    }
}
```

Exemple : balle v1

- problème : l'animation de la balle bloque l'IHM
 - les boutons ne sont plus actifs
 - impossible de fermer la fenêtre
- solution : séparer IHM et balle dans deux threads
 - le thread principal gère l'IHM
 - un nouveau thread est créé pour la balle

Exemple : balle v2

- transformer la balle en thread
 - extension de la classe Thread
 - surcharger la méthode run()
 - démarrer les threads avec start()
 - un thread pour chaque balle

Exemple : balle v2 (1/2)

```
class Ball {  
    private Component canvas;  
    private static final int XSIZE = 15;  
    private static final int YSIZE = 15;  
    private int x = 0;  
    private int y = 0;  
    private int dx = 2;  
    private int dy = 2;  
    public static boolean freeze = false;  
  
    public Ball(Component c) { canvas = c; }  
  
    public void draw(Graphics2D g2) {  
        g2.fill(new Ellipse2D.Double(x, y, XSIZ  
    }  
  
    public void go() {  
        try {  
            while (true) {  
                if (!freeze)  
                    move();  
            }  
        }  
        catch (InterruptedException exception) {}  
    }  
}
```



```
class Ball extends Thread {  
    private Component canvas;  
    private static final int XSIZE = 15;  
    private static final int YSIZE = 15;  
    private int x = 0;  
    private int y = 0;  
    private int dx = 2;  
    private int dy = 2;  
    public static boolean freeze = false;  
  
    public Ball(Component c) { canvas = c; }  
  
    public void draw(Graphics2D g2) {  
        g2.fill(new Ellipse2D.Double(x, y, XSIZ  
    }  
  
    public void run() {  
        try {  
            while (true) {  
                if (!freeze)  
                    move();  
            }  
        }  
        catch (InterruptedException exception) {}  
    }  
}
```

Exemple : balle v2 (2/2)

```
class BallCanvas extends JPanel {  
    private ArrayList balls = new ArrayList();  
  
    public void addBall() {  
        Ball b = new Ball(this);  
        balls.add(b);  
        b.go();  
    }  
  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        Graphics2D g2 = (Graphics2D)g;  
        for (int i = 0; i < balls.size(); i++) {  
            Ball b = (Ball)balls.get(i);  
            b.draw(g2);  
        }  
    }  
}
```

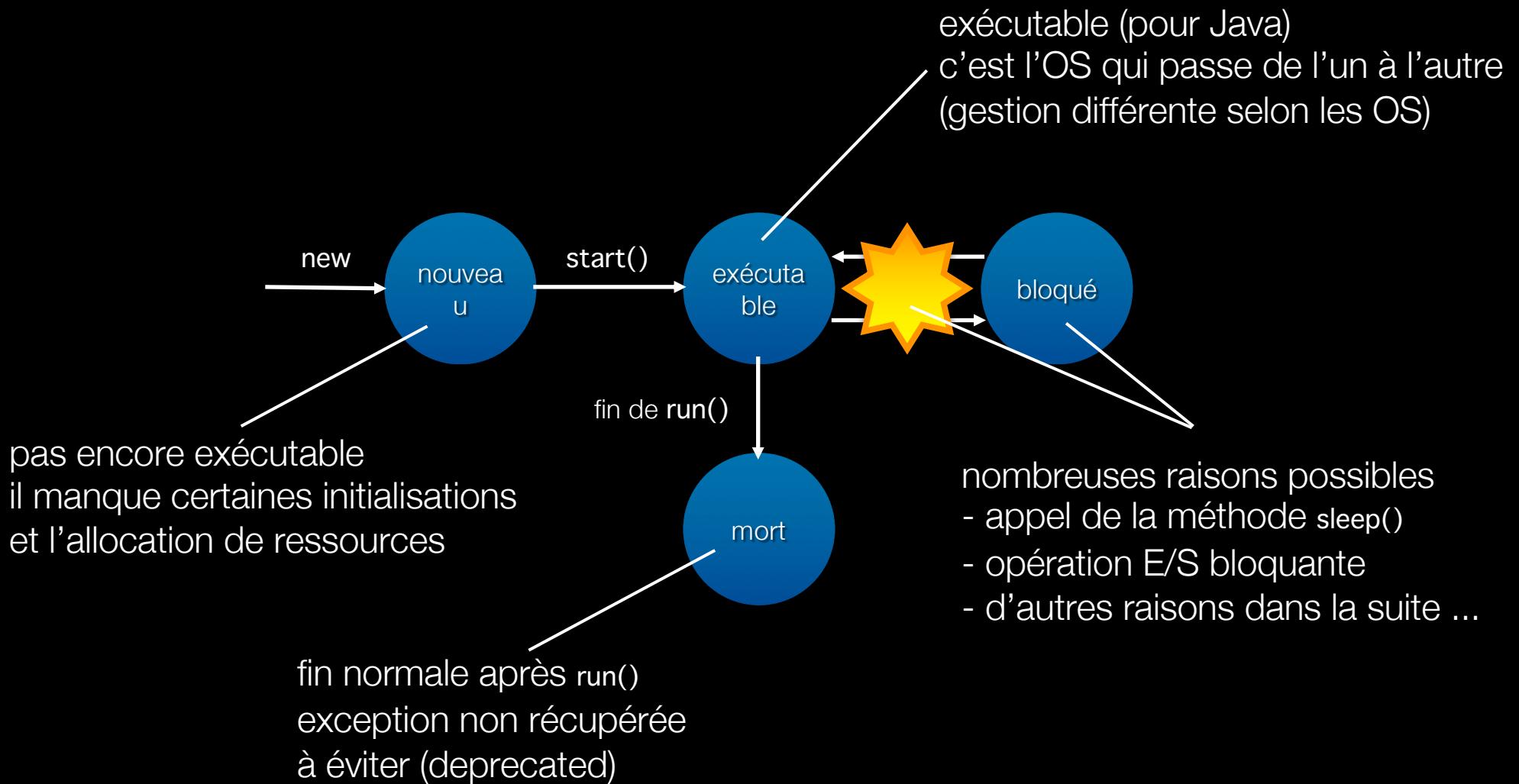


```
class BallCanvas extends JPanel {  
    private ArrayList balls = new ArrayList();  
  
    public void addBall() {  
        Ball b = new Ball(this);  
        balls.add(b);  
        b.start();  
    }  
  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        Graphics2D g2 = (Graphics2D)g;  
        for (int i = 0; i < balls.size(); i++) {  
            Ball b = (Ball)balls.get(i);  
            b.draw(g2);  
        }  
    }  
}
```

Java.lang.Thread

- Thread()
 - nouveau thread
- run()
 - méthode principale du thread, doit être surchargée
- start()
 - lance le thread et appelle run()
 - redonne la main, le nouveau thread est exécuté en parallèle (concurrence)

Etats d'un thread



Interruption de threads (1/3)

- principe :
un thread s'arrête quand sa méthode run() est terminée
- pas de méthode intégrée pour mettre fin au thread
- run() doit vérifier régulièrement si elle doit terminer
- ```
public void run() {
 while (encore du travail) {
 faire le travail;
 se mettre en sommeil;
 }
}
```

# Interruption de threads (2/3)

- problème :  
le thread ne peut pas déterminer s'il doit s'arrêter pendant qu'il dort
- solution :  
utiliser la méthode `interrupt()` et `InterruptedException`
- ```
public void run() {  
    try {  
        while (encore du travail) {  
            faire le travail;  
            se mettre en sommeil;  
        }  
    catch (InterruptedException e) { ... }  
    }  
}
```

Interruption de threads (3/3)

- problème :
pas d'exception levée si l'appel à `interrupt()` se produit pendant que le thread est actif ou bloqué sur une opération d'E/S
- solution :
utiliser la méthode `interrupted()`
- ```
public void run() {
 try {
 while (!interrupted() && encore du travail) {
 faire le travail;
 se mettre en sommeil;
 }
 } catch (InterruptedException e) { ... }
}
```

# Exemple : balle v3 (1/2)

```
class BallCanvas extends JPanel {
 private ArrayList balls = new ArrayList();

 public void addBall() {
 Ball b = new Ball(this);
 balls.add(b);
 b.start();
 }

 public void paintComponent(Graphics g) {
 super.paintComponent(g);
 Graphics2D g2 = (Graphics2D)g;
 for (int i = 0; i < balls.size(); i++) {
 Ball b = (Ball)balls.get(i);
 b.draw(g2);
 }
 }
}
```



```
class BallCanvas extends JPanel {
 private ArrayList balls = new ArrayList();

 public void addBall() {
 Ball b = new Ball(this);
 balls.add(b);
 b.start();
 }

 public void interrupt() {
 if (balls.size() != 0) {
 Ball b = (Ball)balls.get(0);
 b.interrupt();
 balls.remove(0);
 }
 repaint();
 }

 public void paintComponent(Graphics g) {
 super.paintComponent(g);
 Graphics2D g2 = (Graphics2D)g;
 for (int i = 0; i < balls.size(); i++) {
 Ball b = (Ball)balls.get(i);
 b.draw(g2);
 }
 }
}
```

# Exemple : balle v3 (2/2)

```
class Ball extends Thread {
 private Component canvas;
 private static final int XSIZE = 15;
 private static final int YSIZE = 15;
 private int x = 0;
 private int y = 0;
 private int dx = 2;
 private int dy = 2;
 public static boolean freeze = false;

 public Ball(Component c) { canvas = c; }

 public void draw(Graphics2D g2) {
 g2.fill(new Ellipse2D.Double(x, y, XSIZ
 }

 public void run() {
 try {
 while (true) {
 if (!freeze)
 move();
 }
 }
 catch (InterruptedException exception) {}
 }
}
```



```
class Ball extends Thread {
 private Component canvas;
 private static final int XSIZE = 15;
 private static final int YSIZE = 15;
 private int x = 0;
 private int y = 0;
 private int dx = 2;
 private int dy = 2;
 public static boolean freeze = false;

 public Ball(Component c) { canvas = c; }

 public void draw(Graphics2D g2) {
 g2.fill(new Ellipse2D.Double(x, y, XSIZ
 }

 public void run() {
 try {
 while (!interrupted()) {
 if (!freeze)
 move();
 }
 }
 catch (InterruptedException exception) {}
 }
}
```

# Java.lang.Thread

- `void interrupt()`
  - envoie une demande d'interruption à un thread et positionne le «flag» interrompu du thread à true
  - si le thread est bloqué au moment de la demande alors une `InterruptedException` est levée
- `static boolean interrupted()`
  - examine si le thread courant a été interrompu et positionne le «flag» interrompu à false
- `boolean isInterrupted()`
  - examine si un thread a été interrompu mais ne modifie pas le «flag» interrompu

# Groupes de threads

- une application peut reposer sur de nombreux threads
- il est possible de les regrouper pour les manipuler (démarrage, arrêt) de façon simplifiée / groupée
- ```
ThreadGroup group_b = new ThreadGroup("balls");
Balle bi = new Thread(group_b, "ball"); // n fois
group_b.start();
```

Java.lang.ThreadGroup

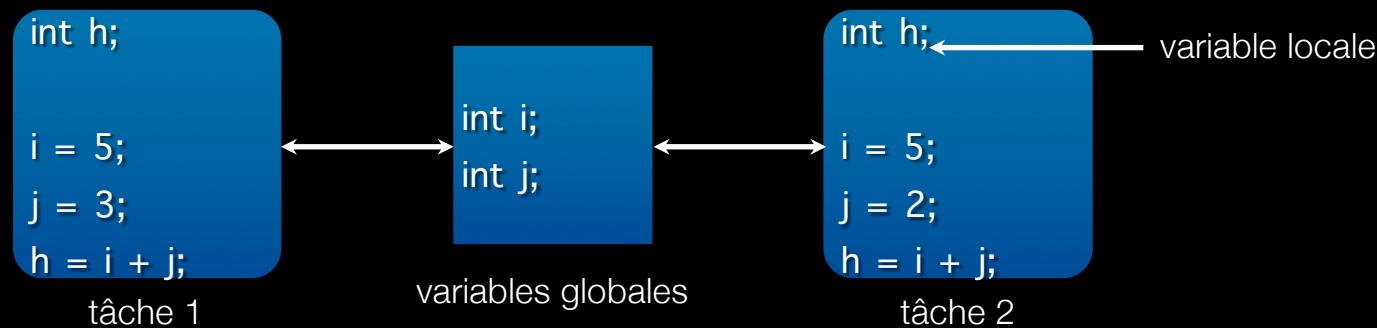
- `ThreadGroup(String name)`
 - nouveau groupe, son parent est le groupe courant
- `ThreadGroup(ThreadGroup parent, String name)`
 - nouveau groupe, son parent est spécifié
 - `int activeCount()` : nb threads actifs dans le groupe
- `ThreadGroup getParent()` : groupe parent
- `void interrupt()` : interruption des threads du groupe ou des groupes fils (et récursivement)

Java.lang.Thread

- Thread(ThreadGroup group, String name)
 - nouveau thread, son groupe est spécifié
- ThreadGroup getThreadGroup()
 - groupe du thread

Problème de ré-entrée (test and set)

- ré-entrée
 - accès concurrent à une ressource
 - partage de données par zone mémoire commune



- test and set
 - problème d'atomicité des opérations
 - `if (solde-retrait > 0) { solde -= retrait; }`

Exemple : Mr & Mrs Smith (1/2)

```
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) {  
        if (somme > 0)  
            if (somme <= valeur) {  
                Thread.currentThread().sleep(50);  
                valeur -= somme;  
                Thread.currentThread().sleep(50);  
                return true;  
            }  
        return false;  
    }  
}
```

```
public class Banque implements Runnable {  
    Compte nom;  
  
    Banque(Compte n) { nom = n; }  
  
    public void Liquide (int montant) {  
        if (nom.retirer(montant)) {  
            Thread.currentThread().sleep(50);  
            Donne(montant);  
            Thread.currentThread().sleep(50);  
        }  
        ImprimeReçu();  
        Thread.currentThread().sleep(50); }  
  
    public void Donne(int montant) {  
        System.out.println(Thread.currentThread().  
            getName()+" : Voici vos " + montant + " euros."); }  
  
    public void ImprimeReçu() {  
        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() {  
        for (int i=1;i<10;i++) {  
            Liquide(100*i);  
            Thread.currentThread().sleep(50)  
        }  
    }  
}
```

Exemple : Mr & Mrs Smith (2/2)

Exécution

```
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();
```

Conseiller Mari: Voici vos 300 euros.
Conseiller Femme: Voici vos 300 euros.
Conseiller Femme: Vous etes fauches!
Conseiller Mari: Vous etes fauches! ...

Mr et Mrs Smith
avaient 1000 €
et ont pu retirer 1200 €

Trace

```
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.
```

Explication ?

- évident si l'on connaît les actions atomiques de chacun des threads (deux ici) et la façon dont le système les exécute
- il faut connaître la sémantique du programme précédent, et donc celle de Java
- prenons un langage jouet

EXPR ::= CONST | VAR | EXPR OPE EXPR

TEST ::= EXPR OPT EXPR | !TEST | TEST && TEST | TEST || TEST

OPE ::= + | - | * | /

OPT ::= < | > | ==

INSTR ::= VAR := EXPR

BLOCK ::= INSTR ; BLOCK | if TEST { BLOCK } { BLOCK } | while TEST { BLOCK }

Sémantique (1/)

- soit L le langage engendré par cette grammaire
- modèle sémantique : systèmes de transitions
- un système de transitions $LTS = (S, S_0, F, E, T)$ où
 - S , est l'ensemble des états possibles du programme
 - $S_0 \in S$, est l'état initial
 - $F \subseteq S$, est un ensemble d'états finaux
 - E , est l'ensemble des étiquettes (événement atomique)
 - $T \subseteq S \times E \times S$, est l'ensemble des transitions

Sémantique (2/)

- que représente un état ?
 - comportement (actions qui restent à faire)
+ mémoire (état des variables du programme)
- soit N l'ensemble des noms de variables à valeur dans \mathbb{Z} ,
un état est un couple (l, H) avec :
 - $l \in L$, comportement
 - $H : V \rightarrow \mathbb{Z}$
avec V l'ensemble des variables du programme

Sémantique (3/)

- que représente une transition ?
 - l'évolution d'un état (I, H) en un état (I', H') en fonction d'une instruction i , dénoté $(I, H) \rightarrow_{\{i\}} (I', H')$
- la définition de la sémantique (opérationnelle) se fait de façon inductive sur la syntaxe de L
- exemples :
 $(x := e; B, H) \rightarrow_{\{x := e\}} (B, H')$ avec
 $H' = H[x \mapsto e(H)]$
 $(\text{if } c \{B1\} \{B2\}; B3, H) \rightarrow_{\{\text{then}\}} (B1; B3, H)$ si $c(H) = \text{true}$
 $(\text{if } c \{B1\} \{B2\}; B3, H) \rightarrow_{\{\text{else}\}} (B2; B3, H)$ si $c(H) = \text{faux}$

Sémantique (4/)

- soit un programme P , sa sémantique $\llbracket P \rrbracket$ est un LTS défini par application des règles sémantiques
- soient n programmes P_i , on peut leur associer à chacun sa sémantique $\llbracket P_i \rrbracket$
- si ces programmes sont en **parallèle**, on note $P_1 \parallel \dots \parallel P_n$
- pour définir la sémantique de $P_1 \parallel \dots \parallel P_n$, on construit $\llbracket P_1 \parallel \dots \parallel P_n \rrbracket$ de façon **compositionnelle** à partir des $\llbracket P_1 \rrbracket \dots \llbracket P_n \rrbracket$

Sémantique (5/)

- les événements d'un processus sont exécutés de façon séquentielle selon la sémantique choisie (LTS)
- sémantique par entrelacement
- étant donnés n LTS $L_i = (S_i, S_{0i}, F_i, E_i, T_i)$,
on définit $L_1 \times \dots \times L_n = (S, s_0, F, E, T)$ avec:
 - $S = S_1 \times \dots \times S_n, s_0 = (s_{01}, \dots, s_{0n}), F = F_1 \times \dots \times F_n,$
 $E = E_1 \cup \dots \cup E_n$
 - $(s_1, \dots, s_i, \dots, s_n) \rightarrow_{\{e\}} (s_1, \dots, s'_i, \dots, s_n) \in T \Leftrightarrow s_i \rightarrow_{\{e\}} s'_i \in T_i$
- $[|P_1|] \dots [|P_n|] = [|P_1|] \times \dots \times [|P_n|]$
- problème : on peut avoir des variables partagées

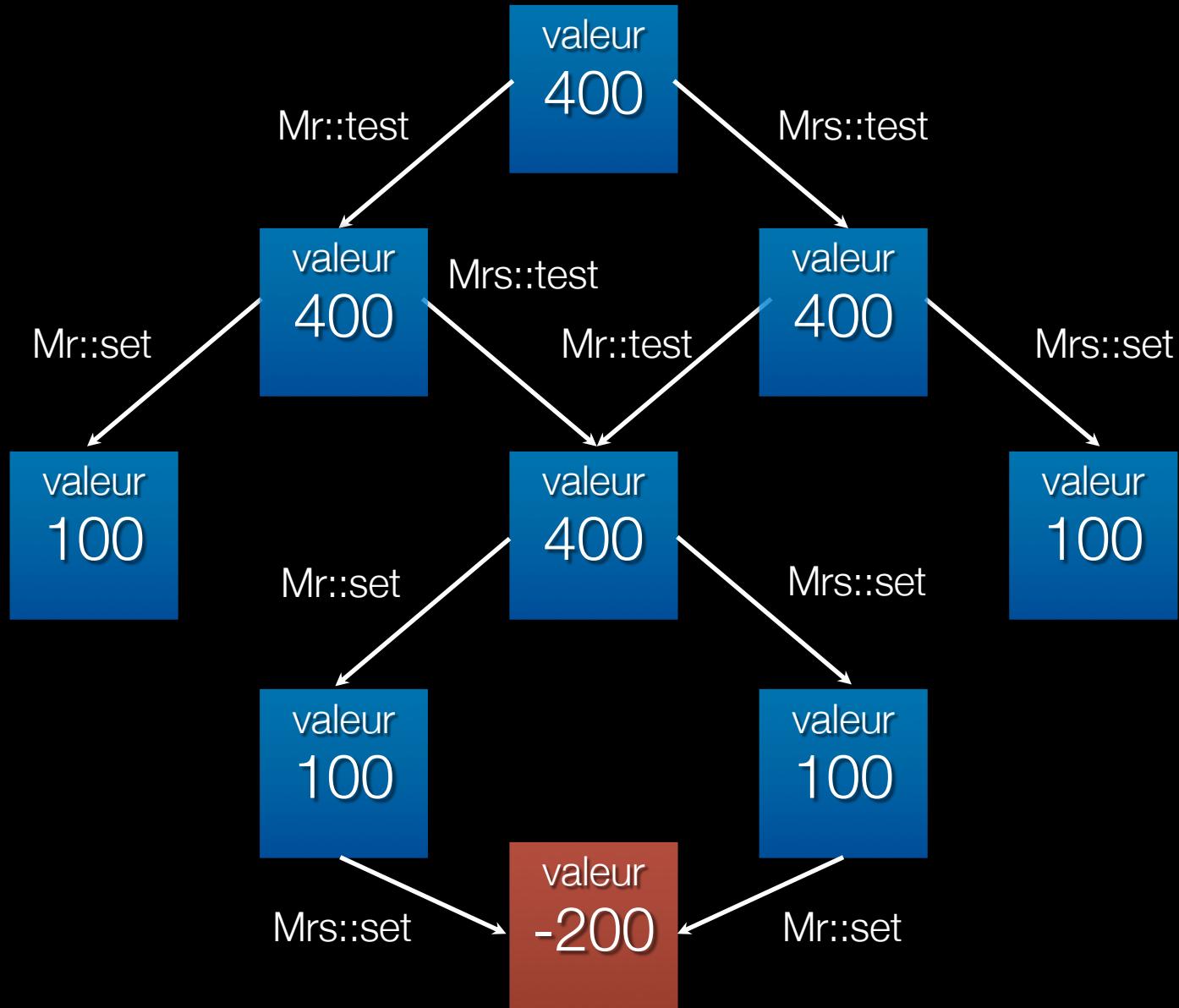
Sémantique (6/)

- on sépare les états pour les variables partagées
- $S = S_1 \times \dots \times S_n \times S_{\text{global}}$
avec $S_{\text{global}} = (\varepsilon, H_{\text{global}})$
- pour les transitions, on perd la compositionnalité
(voir transparent suivant)
- une **exécution** est une trace $s_0 \rightarrow \dots \rightarrow s_f$ avec $s_f \in F$

Sémantique (7/)

- soit $S = (S_1, \dots, S_i, \dots, S_n, S_g)$, avec $\forall i \in \{1, \dots, n\} S_i = (P_i, H_i)$, et $S_g = (\varepsilon, H_g)$. Notons $H_i^+ = H_i \cup H_g$, v^l une variable locale, v^g une variable globale
- $S \rightarrow_{\{a\}} (S_1, \dots, S'_i, \dots, S_n, (\varepsilon, H'_g)) \Leftrightarrow$
 - $P_i = v^l := e; P'_i \quad \wedge a = v^l := e$
 - $\wedge S'_i = (P'_i, H_i[v^l \mapsto e(H_i^+)]) \wedge H'_g = H_g$
- $\vee P_i = v^g := e; P'_i \quad \wedge a = v^g := e$
 - $\wedge S'_i = (P'_i, H_i) \wedge H'_g = H_g[v^g \mapsto e(H_i^+)]$
- $\vee P_i = \text{if } c \{P'_i\} \{P''_i\} \wedge c(H_i^+) = \text{true} \quad \wedge a = \tau_{\text{then}}$
 - $\wedge S'_i = (P'_i, H_i) \wedge H'_g = H_g$
- $\vee P_i = \text{if } c \{P'_i\} \{P''_i\} \wedge c(H_i^+) = \text{false} \wedge a = \tau_{\text{else}}$
 - $\wedge S'_i = (P''_i, H_i) \wedge H'_g = H_g$
- $\vee \dots$

Retour sur Mr & Mrs Smith (1/2)



Retour sur Mr & Mrs Smith (2/2)

- notre langage jouet (et sa sémantique) correspondent-ils à Java ? il faut donc répondre à deux questions :
 - quelles sont les actions atomiques ?
 - comment la JVM traite t-elle les threads ?
(nous avons répondu à cette question, c'est la sémantique par entrelacement à quelques détails près qui sort du cadre du cours)
- JVM
 - chaque thread est chargé en mémoire avec une mémoire locale
 - les variables partagées (ici instance de Compte) sont logées dans la mémoire du thread maître (JVM / main())
 - chaque thread possède une copie locale des variables partagées (cache), la cohérence de la mise à jour au niveau bas étant gérée par la JVM

Modèle mémoire Java théorique

v = v + 1;

read v

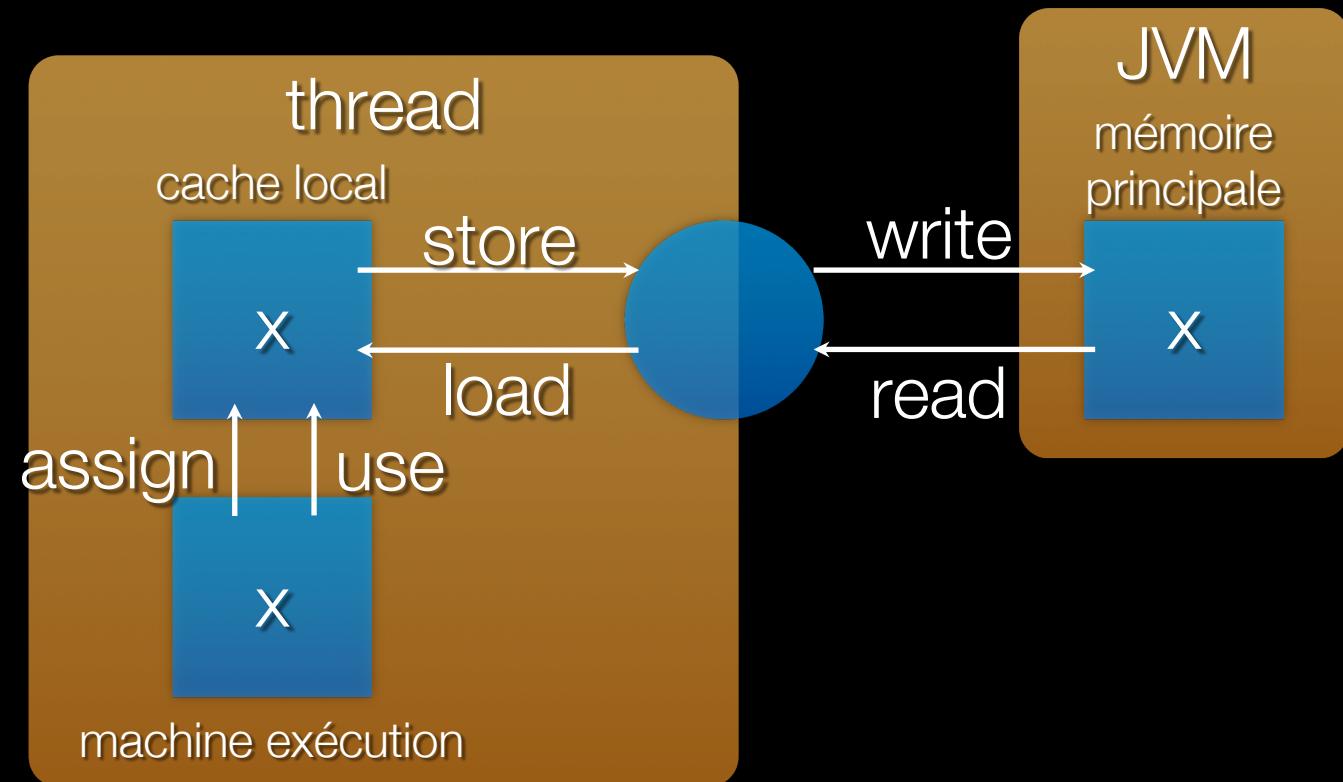
load v

use v (calcul)

assign v

store v

write v



- nb: pour les doubles et les longs ces actions ne sont pas atomiques

synchronized

- Problème :
 - accès à des ressources partagées par plusieurs threads
 - possible conflit test/set, voire corruption de données
- Solution :
 - utilisation du mot-clé **synchronized** en Java
 - pour une méthode : assure que l'ensemble du code de la méthode pour chaque objet ne soit exécuté que par un thread à la fois

Verrous d'objets en Java

- Principe :
(t thread, o objet, **ms** méth. synch., **m** méth. non synch.)

- si **t** appelle **o.ms₁**, alors **o** est verrouillé
(t a le verrou de o)

puis :

- **t** a l'autorisation d'accéder à **o.ms₂**, **o.ms₃**, ...
 - si **t'** appelle **o.ms₂**, alors **t'** est bloqué
(jusqu'à ce que le verrou sur o soit levé)
 - **t'** est libre d'appeler **o.m₁**, **o.m₂**, ...

Mr & Mrs Smith (la solution)

```
public class Compte {  
    private int valeur;  
  
    Compte(int val) { valeur = val; }  
  
    public int solde() {  
        return valeur;  
    }  
  
    public void synchronized depot(int somme) {  
        if (somme > 0)  
            valeur+=somme;  
    }  
  
    public boolean synchronized retirer(int somme) {  
        if (somme > 0)  
            if (somme <= valeur) {  
                Thread.currentThread().sleep(50);  
                valeur -= somme;  
                Thread.currentThread().sleep(50);  
                return true;  
            }  
        return false;  
    }  
}
```

```
public class Banque implements Runnable {  
    Compte nom;  
  
    Banque(Compte n) { nom = n; }  
  
    public void Liquide (int montant) {  
        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() {  
        for (int i=1;i<10;i++) {  
            Liquide(100*i);  
            Thread.currentThread().sleep(50)  
        }  
    }  
}
```

Actions atomiques et JVM

- pour un thread, on rajoute
 - lock (P)
réclame à la mémoire p^{ale} le verrou associé à un objet
 - unlock (V)
libère le verrou associé à un objet
- appel d'une méthode synchronized o.m par t
 - si o non verrouillé, il le devient
 - si o verrouillé, t est bloqué (jusqu'à libération par autre t')

Verrous et sémantique

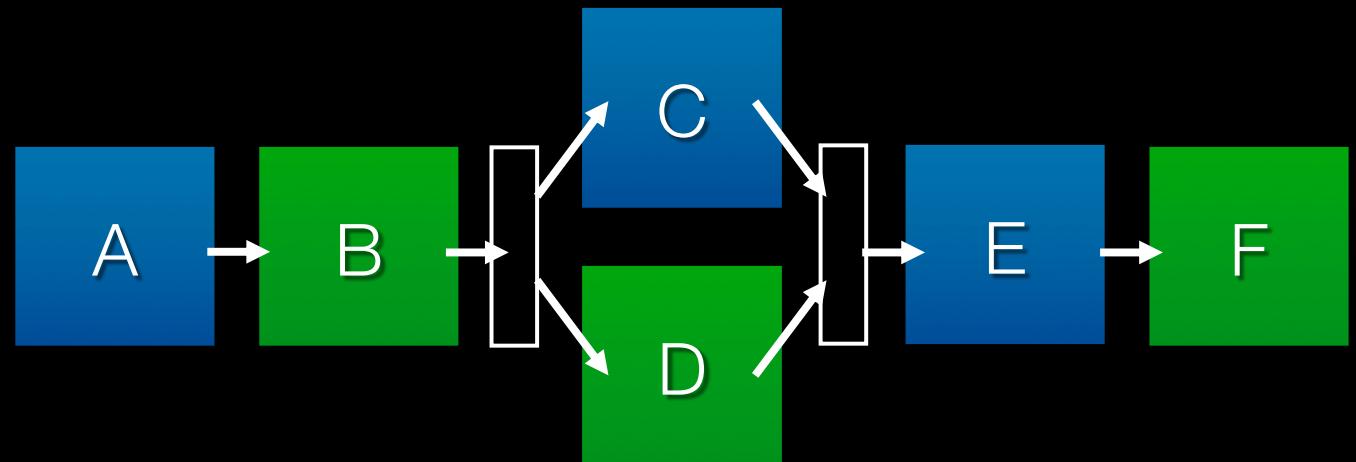
- pour chaque variable v , on rajoute :
 - deux actions, $P(v)$ et $V(v)$
 - une fonction verrou, $k : \text{Var} \rightarrow \{0,1\}$
- on rajoute k aux états, *i.e.*, états : (I, H, k)
- sémantique de $P(v)$, en supposant que v existe
 $(P(v); B, H, k) \rightarrow_{\{P(v)\}} (B, H, k[v \mapsto 0])$ si $k(v)=1$
c-à-d. que si $k(v)=0$ alors le thread est bloqué
- sémantique de $V(v)$, en supposant que v existe
 $(V(v); B, H, k) \rightarrow_{\{V(v)\}} (B, H, k[v \mapsto 1])$
- c'est le blocage des transitions P qui limite l'entrelacement⁴⁵

Remarques sur synchronized

- l'utilisation de synchronized provoque la sérialisation des exécutions
- conséquence : ne les utiliser que lorsque c'est nécessaire, sinon baisse des performances de l'application
- le mécanisme de synchronized est implicite et statique
- il peut être nécessaire d'avoir des mécanismes plus souples, explicites et dynamiques, de verrouillage

Mécanismes explicites de verrouillage / déverrouillage

- deux processus



- P_1 fait {A,C,E}

- P_2 fait {B,D,F}

- on veut le workflow ci-dessus
(en séquence ABCDEF ou ABDCEF)

- quels sont les comportements de P_1 et P_2 ?

wait et notify : principes (1/2)

- nécessité de mettre en place un mécanisme de blocage contrôlé des threads
- **wait**
 - thread courant bloqué, rend le verrou, mis en liste d'attente
- **notify** (resp. **notifyAll**)
 - supprime un (resp. tous les) thread de la liste d'attente
un thread supprimé de la liste redevient exécutable
 - lorsque le verrou est à nouveau disponible l'un des thread le prend et continue son exécution (`InterruptedException`)
- appels possibles si l'appelant a le verrou, sinon exception

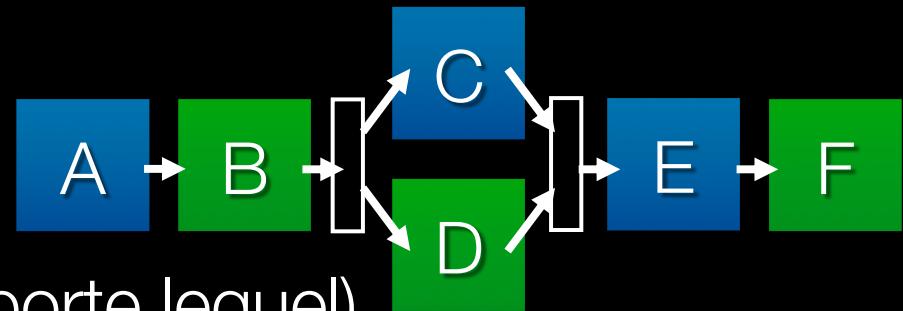
wait et notify : principes (2/2)

- **attention !**
un thread qui appelle `wait` n'a pas moyen de s'auto-réveiller si tous les threads appellent `wait` sans qu'aucun n'appelle `notify` alors on aura un **blocage** (deadlock)
- **attention ! (bis)**
les threads en attente ne sont pas réactivés si aucun thread ne travaille sur l'objet
- lors d'un appel à `notify`, impossible de savoir le thread qui sera débloqué - éventuellement utiliser `notifyAll`
- utiliser `notifyAll` lorsqu'un changement d'état d'objet pourraît être avantageux pour d'autres threads

wait et notify : vérification

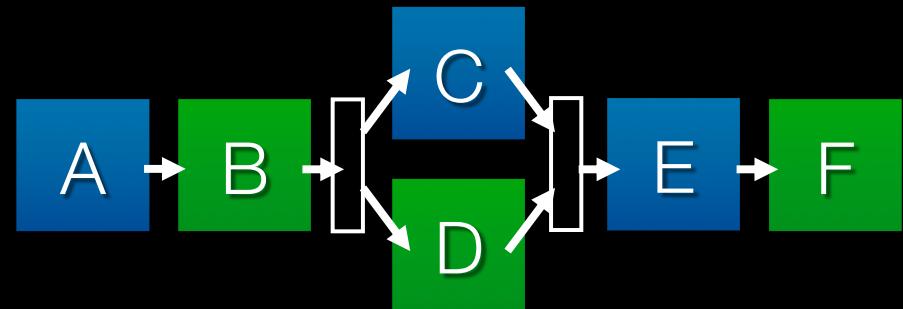
- version simple pour 2 processus et 1 verrou
- on note état (P_x) avec P_x programme
 $[P_x]$ donné par les évolutions de $\wedge; P_x$ (\wedge dénote le démarrage)
- six règles de calcul de $[P_1 \parallel P_2]$
- R1 : si $P_1 \rightarrow_{\{E\}} P_1' \wedge E \notin \{N,W\}$
alors $(P_1, P_2) \rightarrow_{\{E\}} (P_1', P_2)$ *[idem R1bis avec $P_2 \rightarrow$]*
- R2 : si $P_1 \rightarrow_{\{N\}} P_1' \wedge \neg P_2 \rightarrow_{\{W\}}$
alors $(P_1, P_2) \rightarrow_{\{\tau-N\}} (P_1', P_2)$ *[idem R2bis ...]*
- R3 : si $P_1 \rightarrow_{\{N\}} P_1' \wedge P_2 \rightarrow_{\{W\}} P_2'$
alors $(P_1, P_2) \rightarrow_{\{\tau-WN\}} (P_1', P_2')$ *[idem R3 bis ...]*

Solution des 2 processus ?



- on a un objet partagé o (ici n'importe lequel)
o.W, resp. o.N, est un raccourci pour une méthode de o qui fait un wait, resp. un notify
- cette solution est-elle correcte ?
 - $P_1 = A(); o.N(); o.W(); C(); E(); o.N();$
 $P_2 = o.W(); B(); o.N(); D(); o.W(); F();$
- et celle-ci ?
 - $P_1 = A(); o.N(); o.W(); C(); \underline{o.W();}; E(); o.N();$
 $P_2 = o.W(); B(); o.N(); D(); \underline{o.N();}; o.W(); F();$

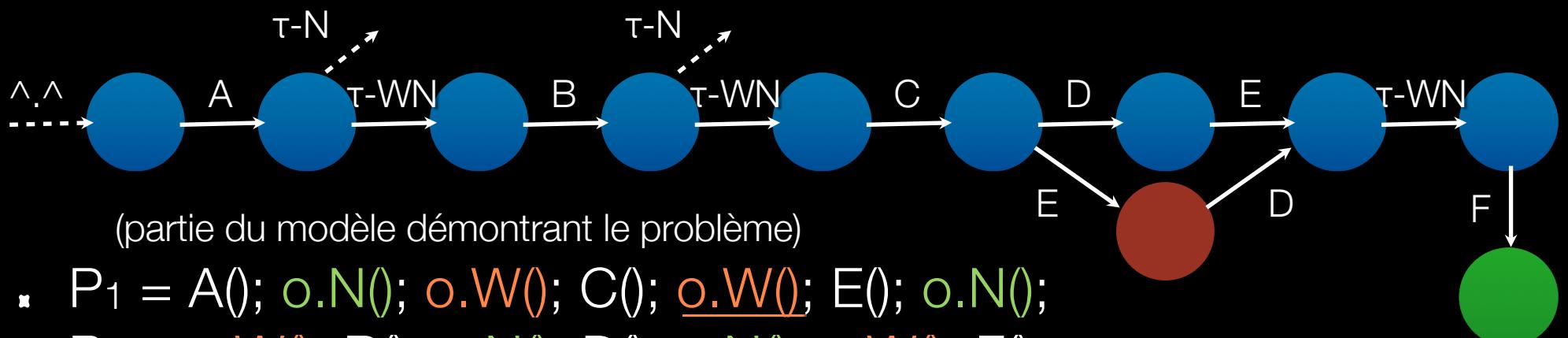
Solution des 2 processus ?



- cette solution est-elle correcte ?
 - $P_1 = A(); \text{o.N(); o.W(); C(); E(); o.N();}$
 $P_2 = o.W(); B(); \text{o.N(); D(); o.W(); F();}$
 - non : on peut avoir ABCEDF (exercice)
- et celle-ci ?
 - $P_1 = A(); \text{o.N(); o.W(); C(); o.W(); E(); o.N();}$
 $P_2 = o.W(); B(); \text{o.N(); D(); o.N(); o.W(); F();}$
 - non : on peut avoir A<blocage> (exercice)

Solution des 2 processus ?

- $P_1 = A(); o.N(); o.W(); C(); E(); o.N();$
 $P_2 = o.W(); B(); o.N(); D(); o.W(); F();$



- $P_1 = A(); o.N(); o.W(); C(); o.W(); E(); o.N();$
 $P_2 = o.W(); B(); o.N(); D(); o.N(); o.W(); F();$



Solution des 2 processus ?

P₁ = A(); o.N(); o.W(); C(); o.W(); E(); o.N();

P₂ = o.W(); B(); o.N(); D(); o.N(); o.W(); F();

```
class Controller {  
    public Controller() {}  
    //  
    public synchronized notify() {  
        notify();  
    }  
    ... // pareil pour B, D et E  
    //  
    public synchronized wait() {  
        try { wait(); }  
        catch (InterruptedException e) {}  
    }  
}  
  
main (...) {  
    Controller c = new Controller();  
    P1 p1 = new P1(c); P2 p2 = new P2(c);  
    p1.start(); p2.start();  
}
```

```
class P1 extends Thread {
```

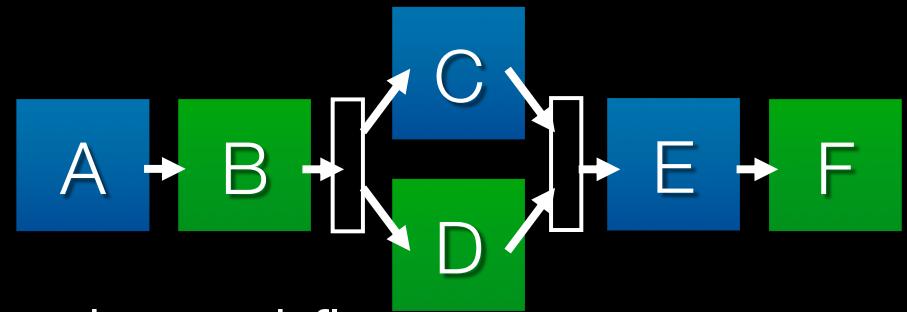
```
    Controller c;  
    public P1(Controller cx) { c=cx; }  
    public void A(...) { ... }  
    ... // definitions de C et E  
    //  
    public void run() {  
        yield();  
        A(...); c.notify();  
        c.wait(); C(...);  
        c.wait(); E(...); c.notify();  
    }  
}  
  
class P2 extends Thread {  
    Controller c;  
    public P1(Controller cx) { c=cx; }  
    public void A(...) { ... }  
    ... // definition de D et F  
    //  
    public void run() {  
        c.wait(); B(); c.notify();  
        D(); yield(); c.notify();  
        c.wait(); F();  
    }  
}
```

Sémaphore en Java

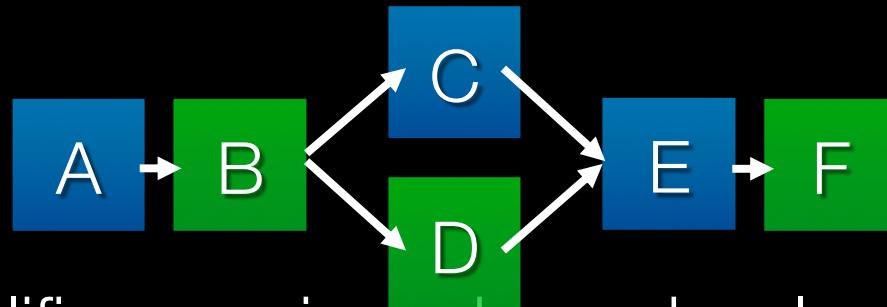
- principe :
 - section critique (SC)
besoin d'exclusion mutuelle
 - un jeton
celui qui l'a peut entrer en SC
les autres bloquent
- implantation de haut niveau de P(.)
et V(.) sur une variable partagée,
ici le jeton (ou sémaphore)

```
public class Semaphore {  
    int n; // jeton  
  
    public Semaphore() { n=1; }  
  
    public synchronized void P() {  
        if (n == 0) {  
            try {  
                wait();  
            } catch(InterruptedException ex) {};  
        }  
        n--;  
        System.out.println("P(jeton)");  
    }  
  
    public synchronized void V() {  
        n=1;  
        System.out.println("V(jeton)");  
        notify();  
    }  
}
```

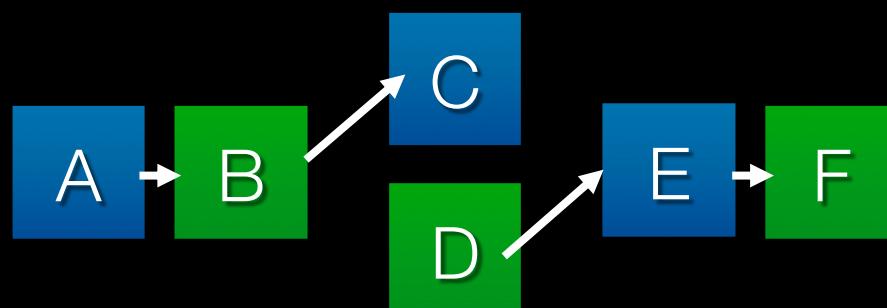
Solution des 2 processus (1/3)



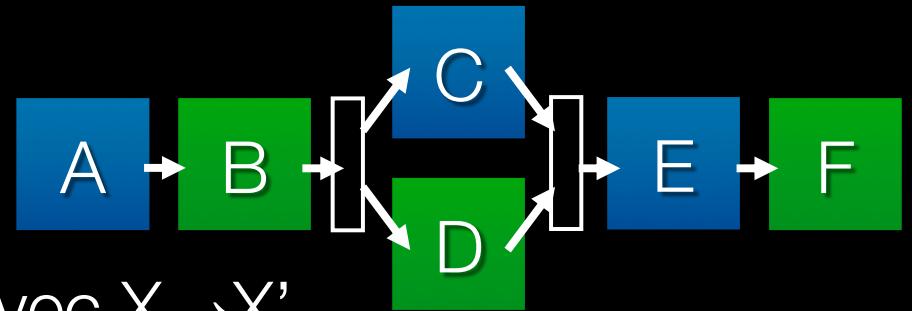
- définissons la structure causale du workflow



- on peut simplifier en raison des ordres locaux



Solution des 2 processus (2/3)

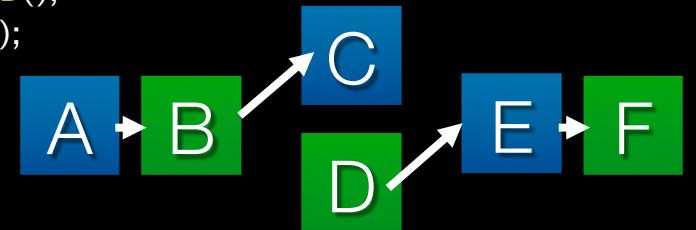


- pour chaque X tq il existe X' avec $X \rightarrow X'$,
on définit un couple de méthodes de contrôle :
 - `doneX() { isDoneX := true ; notify(); }`
 - `waitX() {if (! isDoneX) { try { wait(); } catch (...) {} }}`
- pour chaque processus P ,
pour chaque action Y de P tq $X \rightarrow Y$ (X action de P')
`... c.waitX(); Y(); c.doneY(); ...`
- c objet contrôleur, référencé dans P et P'

Solution des 2 processus (3/3)

```
class Controller {  
    bool isDoneA, isDoneB, isDoneD, isDoneE;  
    public Controller() {  
        isDoneA=false; isDoneB=false;  
        isDoneD=false; isDoneE=false;  
    }  
    //  
    public synchronized doneA() {  
        isDoneA=true;  
        notify();  
    }  
    ... // pareil pour B, D et E  
    //  
    public synchronized waitA() {  
        if (! isDoneA) {  
            try { wait(); }  
            catch (InterruptedException e) {}  
        }  
        .... // pareil pour B, D et E  
    }  
  
    main (...) {  
        Controller c = new Controller();  
        P1 p1 = new P1(c); P2 p2 = new P2(c);  
        p1.start(); p2.start();  
    }  
    class P1 extends Thread {
```

```
        Controller c;  
        public P1(Controller cx) { c=cx; }  
        public void A(...) { ... }  
        ... // definitions de C et E  
        //  
        public void run() {  
            A(...); c.doneA();  
            c.waitB(); C(...);  
            c.waitD(); E(...); c.doneE();  
        }  
    }  
  
    class P2 extends Thread {  
        Controller c;  
        public P1(Controller cx) { c=cx; }  
        public void A(...) { ... }  
        ... // definition de D et F  
        //  
        public void run() {  
            c.waitA(); B(); c.doneB();  
            D(); c.doneD();  
            c.wait(); F();  
        }  
    }
```



Solution ? vraiment ?

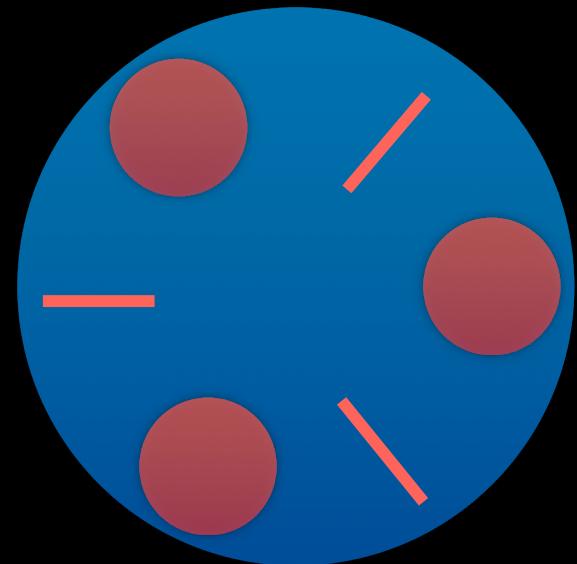
- on peut reprendre l'idée de base de la vérification avec 2 processus et 1 verrou mais remarques :
 - les `wait()` sont faits conditionnellement p/r à une variable et avant `notify()` on modifie la valeur de la variable concernée
 - les `synchronized` nous simplifient le travail (atomicité test-wait et set-notify)
- modifications nécessaires :
 - prendre en compte un certain nombre de variables (les `isDonexx`) dans l'état global, dans `[|Px|]` et dans `[| P1 || P2 |]`
 - exercice : proposer une technique pour vérifier la solution

Remarque

- ici le cas est simple : 2 processus
- dans le cas général, quand on se fait réveiller on n'est pas forcément sur que l'objet est dans l'état qui nous arrange
- pour chaque X tq il existe X' avec $X \rightarrow X'$,
on définit un couple de méthodes de contrôle :
 - `doneX() { isDoneX := true ; notifyAll(); }`
 - `waitX() {while (! isDoneX) { try { wait(); } catch (...) {} }}`
- pour chaque processus P ,
pour chaque action Y de P tq $\{X_1, \dots, X_n\} \rightarrow Y$
 $\dots \underline{c.waitX_1(); \dots; c.waitX_n(); Y(); c.doneY(); \dots}$

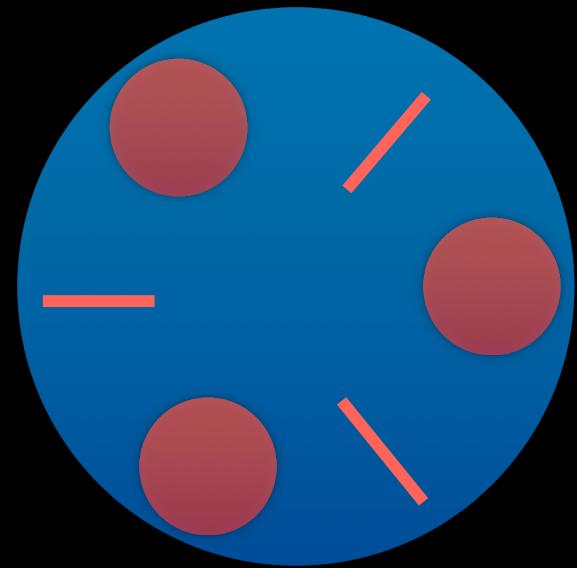
Un classique ! (les philosophes)

- principe :
 - n philosophes (P_1, \dots, P_n)
 - table ronde, un fourchette entre chaque philosophe
 - algorithme d'un philosophe :
 - prendre f. gauche puis droite
 - manger
 - rendre f. gauche puis droite



Un classique ! (les philosophes)

- exercice :
- utilisez les sémaphores pour programmer le dîner de trois philosophes : Platon, Descartes et Spinoza
- un philosophe a un nom ainsi que les deux fourchettes qui l'entourent
- chaque fourchette est un sémaphore (jeton)



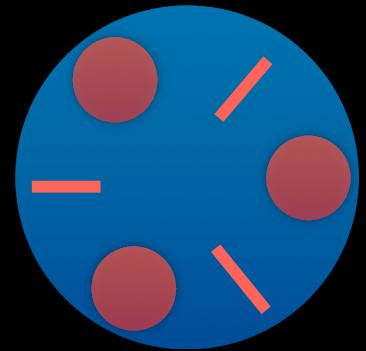
Philosophes : la solution ?

```
public class Fork {  
    int n;  
    String name;  
    public Fork(String x) { n=1; name=x; }  
    public synchronized void P( ) {  
        if (n == 0) {  
            try {  
                wait();  
            } catch(InterruptedException ex) {};  
        }  
        n--;  
        System.out.println("P("+name+");");  
    }  
    public synchronized void V() {  
        n=1;  
        System.out.println("V("+name+");");  
        notify();  
    }  
}  
  
public static void main(...) {  
    Fork a = new Fork("a"); Forkb = new Fork("b");  
    Fork c = new Fork("c");  
    Phil Phil1 = new Phil(a,b); Phil Phil2 = new  
    Phil(b,c); Phil Phil3 = new Phil(c,a);  
    Phil1.setName("Spinoza");Phil2.setName("Descartes")  
}
```

```
    Phil3.setName("Platon");  
    Phil1.start(); Phil2.start(); Phil3.start();  
}  
public class Phil extends Thread {  
    Fork LeftFork;  
    Fork RightFork;  
    String name;  
    public Phil(Fork l, Fork r) {  
        LeftFork = l; RightFork = r;  
    }  
    public void setName(String n) {  
        name = n;  
    }  
    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) {};  
    }  
}
```

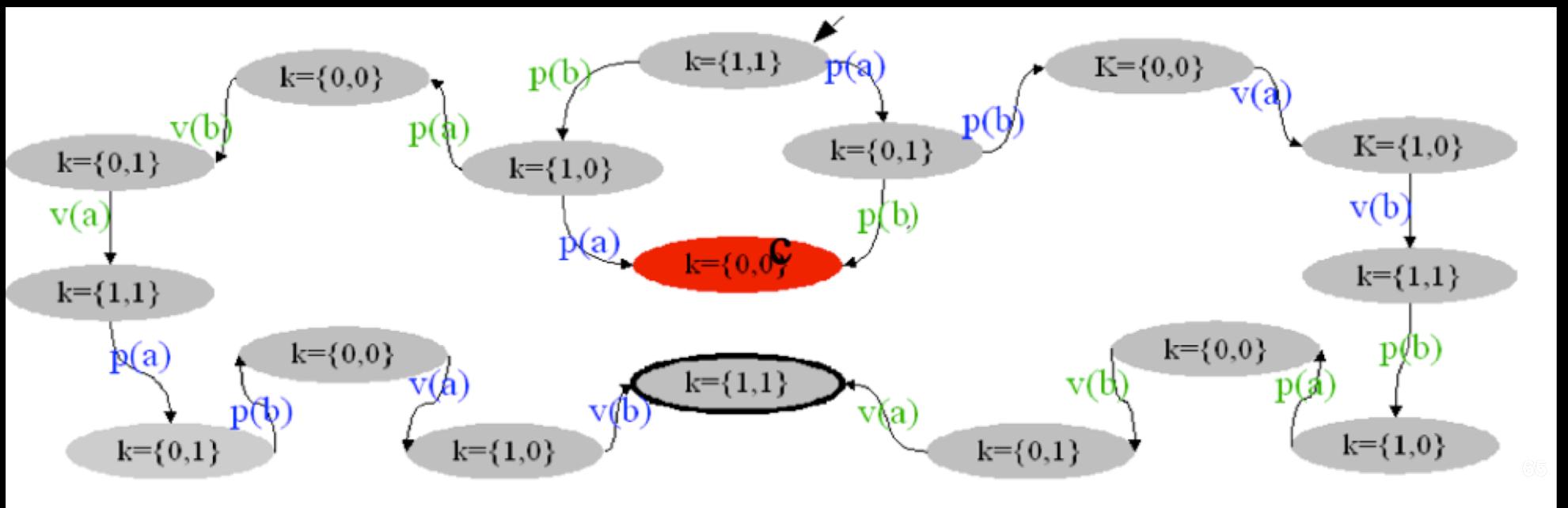
Les philosophes : le problème

- intuition : si chaque philosophe prend sa fourchette gauche, tout le monde est bloqué (sur wait)
- notions de **blocages** - le système n'avance pas vers la terminaison car :
 - blocage mort (**deadlock**)
aucune action possible
 - blocage vivant (**livelock**)
suites d'actions possibles ramènent à un état déjà parcouru (existance de cycles) [version forte ou faible]



Les philosophes : le problème

- utilisons notre modèle (sur un cas simple)
- $P_1 = P(a); P(b); V(a); V(b);$
 $P_2 = P(b); P(a); V(b); V(a);$
- cas de parallélisme/entrelacement, $P_1 \parallel P_2$
(on représente juste k)

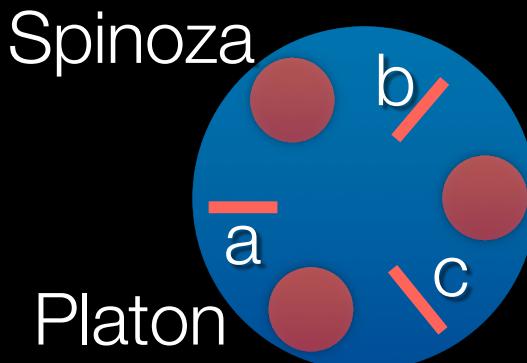


Conditions au blocage mort

- Coffman, Elphick et Shoshani (1971) ont identifié quatre conditions nécessaires et suffisantes pour qu'apparaissent des problèmes de deadlock :
 - partage de ressource utilisée sous une exclusion mutuelle
 - acquisition incrémentale de ressource ressource demandée avant d'en libérer une autre
 - non-préemption de ressource pas de temps limite à l'acquisition d'une ressource
 - circuit dans la dépendance des ressources entre processus

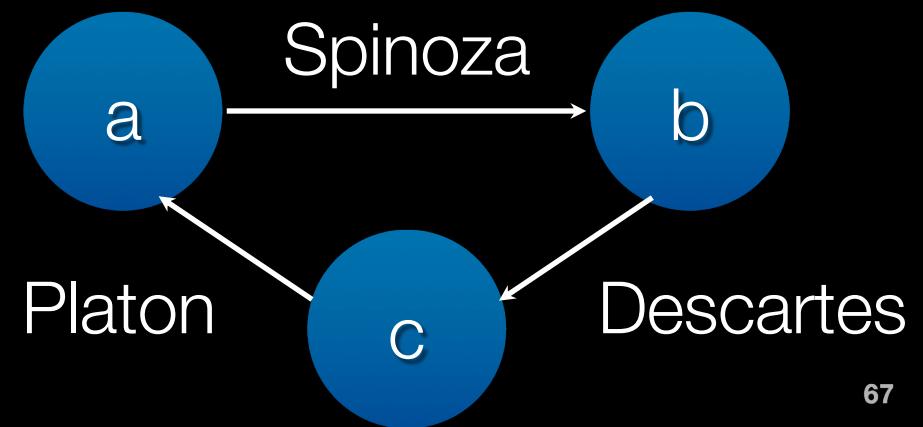
Compléments sur le blocage

- le problème de blocage vient d'un ensemble de besoins ordonnés localement et circulaires globalement
- graphe des requêtes sur les ressources
 - un noeud par ressource
 - arc $n \rightarrow m$ ssi il existe un processus qui, ayant acquis le verrou sur n , demande sans l'avoir relâché un verrou sur m



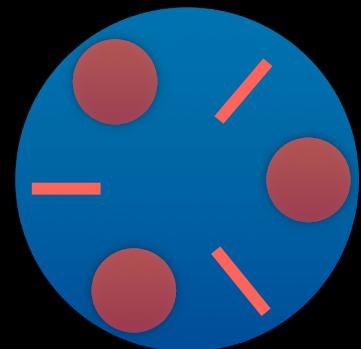
Descartes

Platon



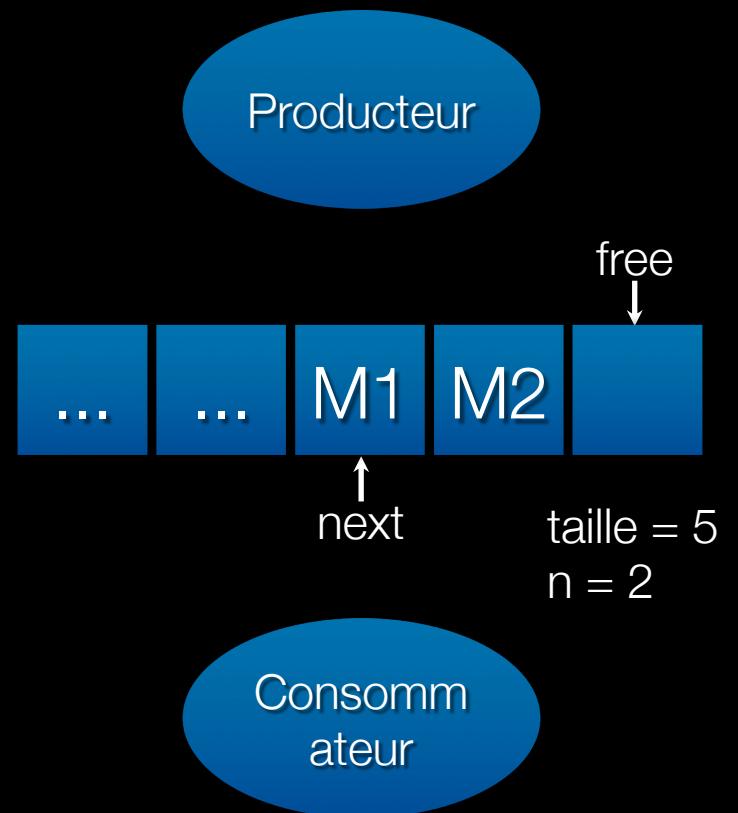
Les philosophes : la solution

- comment corriger l'application sachant qu'on désire garder la symétrie de l'algorithme (propriété désirée facilitant l'analyse) ?
- garder le même algorithme, jouer sur la position :
 - paire : prendre gauche puis droite
 - impaire : prendre droite puis gauche
- exercice : donnez le graphe de requête et preuve de non blocage



Producteur-Consommateur

- autre exemple classique
- zone tampon Z de taille finie (n)
- le producteur produit une donnée et la met dans Z
le consommateur prend une donnée dans Z et la consomme
- contraintes :
 - ne pas lire et écrire en même temps
 - ne pas écrire si Z plein
 - ne pas lire si Z vide
- utilisation d'un sémaphore à compteur

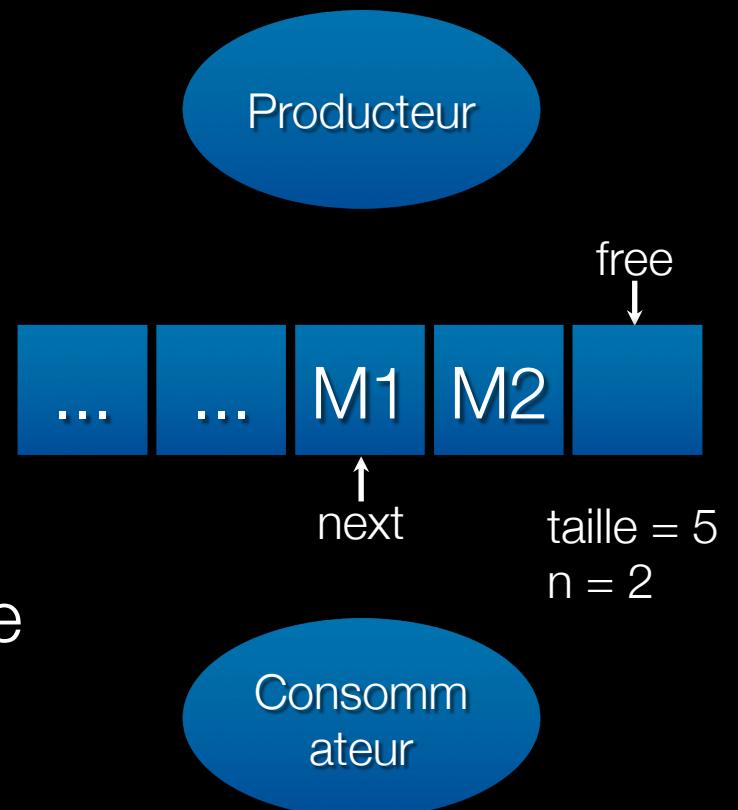


Sémaphore à compteur

```
public class Semaphore {  
    int n;  
    int max;  
  
    public Semaphore(int nb) {  
        max=nb;  
        n = max;  
    }  
  
    public synchronized void P() {  
        if (n == 0) {  
            try {  
                wait();  
            } catch(InterruptedException ex) {};  
        }  
        n--;  
        System.out.println("P("+max-n+"ieme)");  
    }  
  
    public synchronized void V( ) {  
        if(n<max) {  
            n++  
            System.out.println("V("+n+"ième)");  
            notify();  
        }  
    }  
}
```

Producteur-Consommateur

- exercice :
 - utilisez les sémaphores à compteur pour implanter le système producteur-consommateur
 - le producteur produit 10x plus vite que le consommateur ne consomme
 - écrivez le code du producteur, celui du consommateur et une main



Producteur-Consommateur

```
public class Consumer extends Thread {
    Tampon zone;
    public Consumer(Tampon t) { zone=t; }

    public void run() {
        while (true) {
            zone.consomme();
            try { this.sleep(5000); }
            catch (InterruptedException e) {};
        }
    }
}

public class Producer extends Thread {
    Tampon zone;
    public Producer(Tampon t) { zone=t; }

    public void run() {
        int i = 0;
        while (true) {
            zone.produit("message "+i);
            try { this.sleep(500); }
            catch (InterruptedException e) {};
            i++;
        }
    }
}

public static void main (...) {
    Tampon t = new Tampon(5);
    int taille; int free=0; int n; int next=0;
    Object [] values ;
    public Tampon(int max) {
        n = max; taille=max; values=new Object[n]; }
    public synchronized void produit(Object b) {
        if (n == 0) {
            try { wait(); }
            catch (InterruptedException ex) {};
        }
        values[free]=b; free=(free+1)%taille;
        n--; System.out.println("P("+b+ ")");
        notify();
    }
    public synchronized Object consomme() {
        if (n == taille) {
            try { wait(); }
            catch (InterruptedException ex) {};
        }
        object o = values[next];
        next=(next+1)%taille;
        n++; System.out.println("P("+o+ ")");
    }
}
```

Communication entre threads

- utilisation de
 - PipedInputStream, PipedOutputStream
 - PipedReader, PipedWriter
- avantage : communication entre threads sans se préoccuper de leur synchronisation (quand variable(s) partagées)
- application : producteur-consommateur (à nouveau, mais un peu plus évolué)

Communication entre threads

- `java.lang.PipedInputStream`
 - `PipedInputStream()`
nouveau flux d'entrée (non connecté)
 - `connect(PipedOutputStream out)`
connexion à un flux de sortie out pour la lecture de données
 - `PipedInputStream(PipedOutputStream out)`
nouveau flux d'entrée connecté à out

Communication entre threads

- `java.lang.PipedOutputStream`
 - `PipedInputStream()`
nouveau flux de sortie (non connecté)
 - `connect(PipedInputStream in)`
connexion à un flux d'entrée in pour l'écriture de données
 - `PipedOutputStream(PipedInputStream in)`
nouveau flux de sortie connecté à in

Communication entre threads

- architecture (nous verrons les ADL plus tard)
 - trois «composants»
 - le producteur génère des nombres
 - le filtre calcule la moyenne
 - le consommateur affiche les moyennes (si $\delta > \epsilon$.)
 - deux «connecteurs» : pipe (style «pipe-and-filter»)



Communication entre threads

```
class Producer extends Thread {  
    private DataOutputStream out;  
    private Random rand = new Random();  
    public Producer(OutputStream os) {  
        out = new DataOutputStream(os); }  
  
    public void run() {  
        while (true) {  
            try {  
                double num = rand.nextDouble();  
                out.writeDouble(num);  
                out.flush();  
  
                sleep(Math.abs(rand.nextInt()%1000));  
            }  
            catch(Exception e) {  
                System.out.println("Error: " + e); }  
        }  
    }  
}  
  
class Consumer extends Thread {  
    private double oldx = 0;  
    private DataInputStream in;  
    private static final double EPS = 0.01;  
    public Consumer(InputStream is) {  
        in = new DataInputStream(is); }  
  
    public void run() {  
        for(;;) {  
            try {  
                double x = in.readDouble();  
                if (Math.abs(x - oldx) > EPS) {  
                    System.out.println(x);  
                    oldx = x;  
                }  
            }  
            catch(IOException e) {  
                System.out.println("Error: " + e); }  
        }  
    }  
}
```

Communication entre threads

```
class Filter extends Thread {  
    private DataInputStream in;  
    private DataOutputStream out;  
    private double total = 0;  
    private int count = 0;  
  
    public Filter(InputStream is, OutputStream os) {  
        in = new DataInputStream(is);  
        out = new DataOutputStream(os);  
    }  
  
    public void run() {  
        for (;;) {  
            try {  
                double x = in.readDouble();  
                total += x; count++;  
                if (count!=0) out.writeDouble(total/count);  
            }  
            catch(IOException e) {  
                System.out.println("Error: " + e);  
            }  
        }  
    }  
}
```

```
import java.util.*;  
import java.io.*;  
  
public class PipeTest {  
    public static void main(String args[]) {  
        try {  
            PipedOutputStream pout1 = \  
                new PipedOutputStream();  
            PipedInputStream pin1 = \  
                new PipedInputStream(pout1);  
            PipedOutputStream pout2 = \  
                new PipedOutputStream();  
            PipedInputStream pin2 = \  
                new PipedInputStream(pout2);  
  
            Producer prod = new Producer(pout1);  
            Filter filt = new Filter(pin1, pout2);  
            Consumer cons = new Consumer(pin2);  
  
            prod.start(); filt.start(); cons.start();  
        }  
    }  
}
```