



# Systèmes et Applications Distribués

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

Bachir Djafri ([bachir.djafri@univ-evry.fr](mailto:bachir.djafri@univ-evry.fr))

Tarek Melliti ([tarek.melliti@univ-evry.fr](mailto:tarek.melliti@univ-evry.fr))

Pascal Poizat ([pascal.poizat@lri.fr](mailto:pascal.poizat@lri.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/Opérations distribués (RMI, CORBA & gRPC)

## **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 implémente 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 + XSIZE >= canvas.getWidth()) {  
            x = canvas.getWidth() - XSIZE;  
            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);  
        }  
    }  
}
```

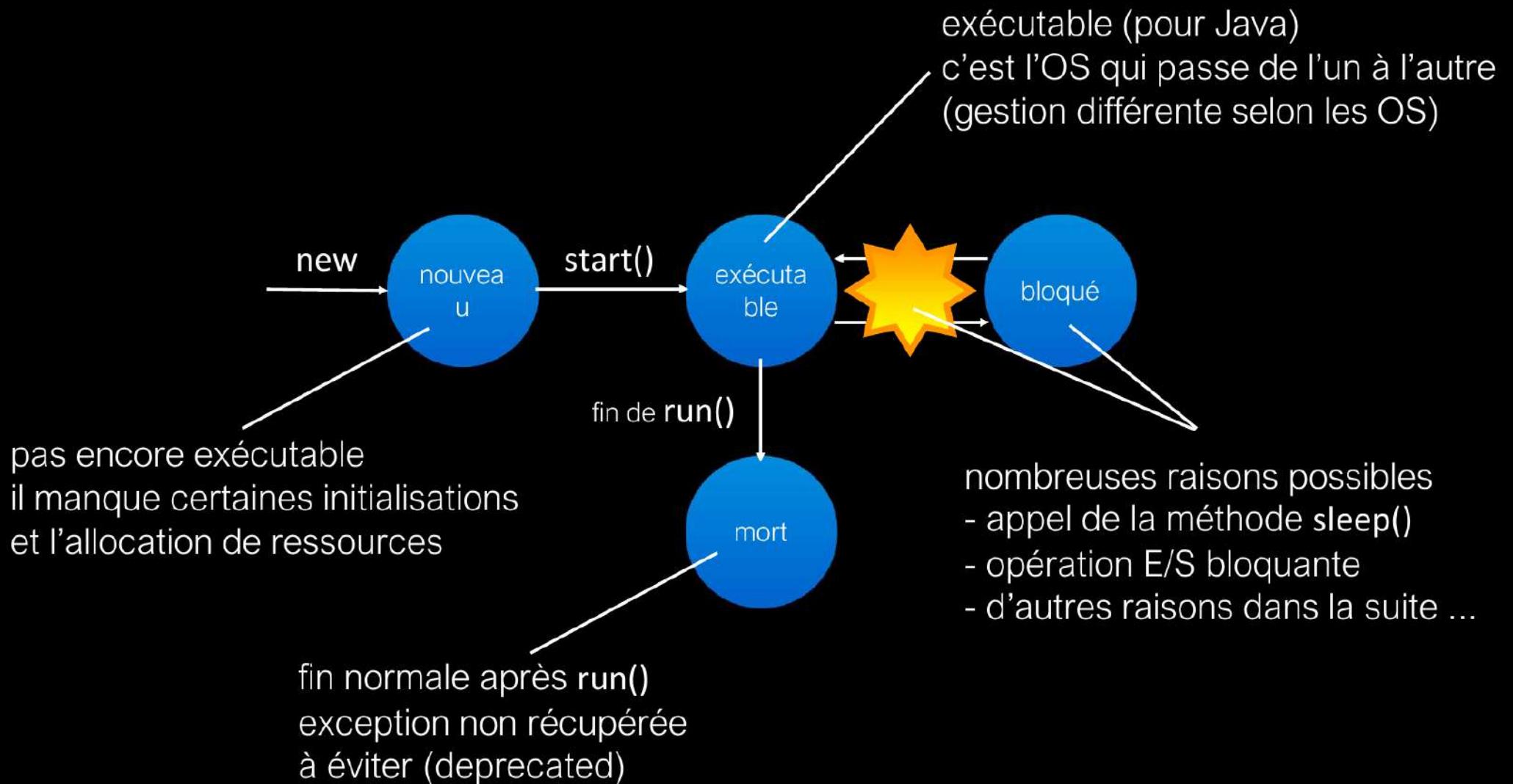


```
class BallCanvas extends JPanel {  
    private ArrayList balls = new ArrayList();  
  
    public void addBall() {  
        Ball b = new Ball(this);  
        balls.add(b);  
        b.start();  
    }  
  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        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

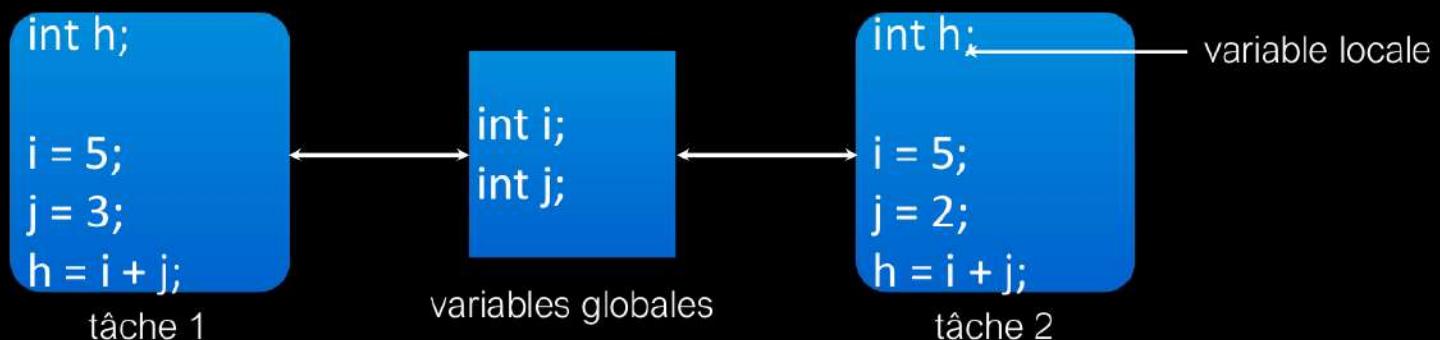
- `ThreadGroup(String name)`
  - nouveau groupe, son parent est le groupe courant
- `ThreadGroup(ThreadGroup parent, String name)`
  - nouveau groupe, son parent est spécifié
- `int activeCount()` : nb threads actifs dans le groupe
- `ThreadGroup getParent()` : groupe parent
- `void interrupt()` : interruption 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 (**s**olde-retrait > 0) { **s**olde -= 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();
```

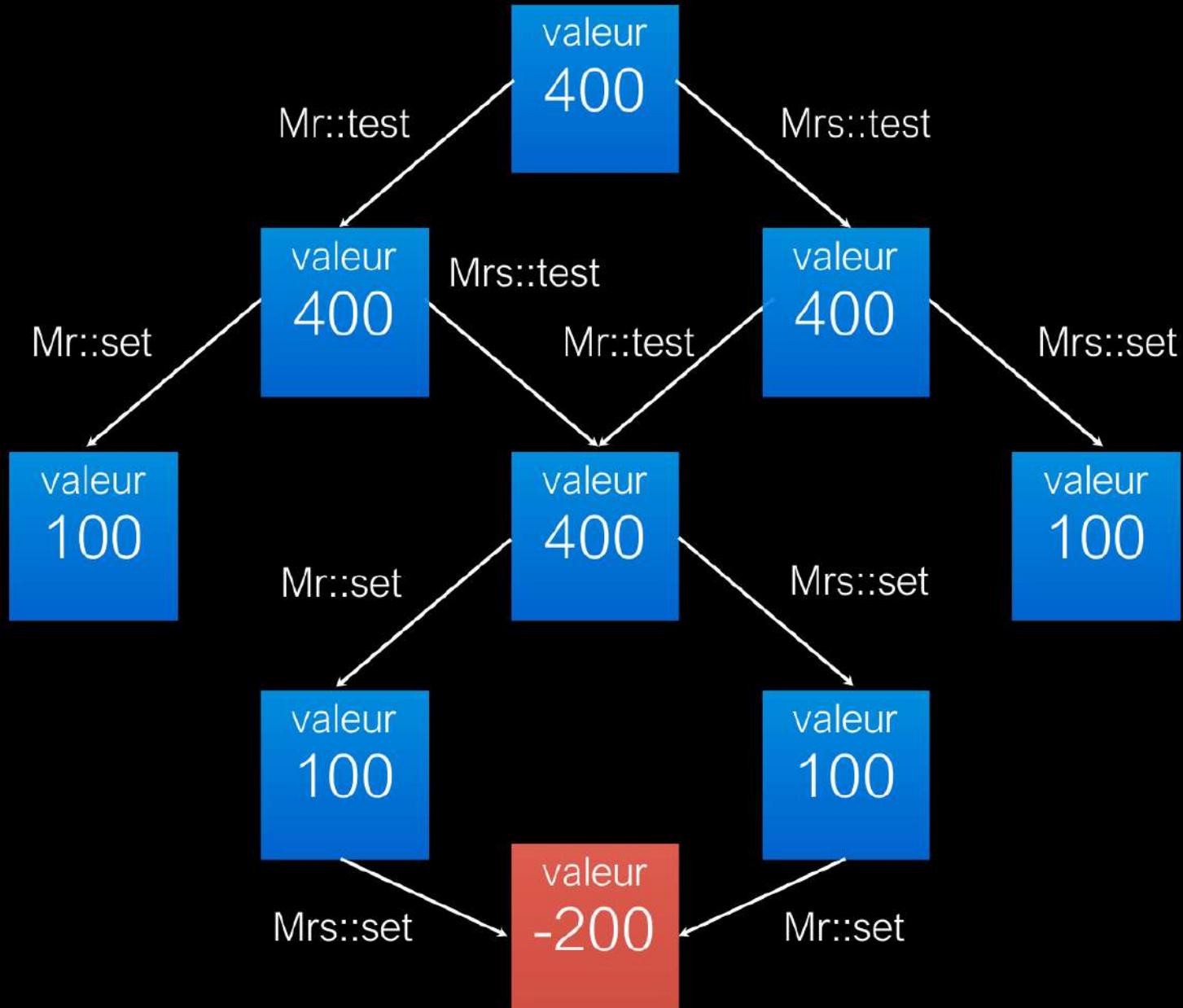
Conseiller Mari: Il vous reste 400 euros.  
Conseiller Mari: Voici vos 300 euros.  
Conseiller Femme: Voici vos 300 euros.  
Conseiller Femme: Vous etes fauches!  
Conseiller Mari: Vous etes fauches! ...

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.
```

# 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

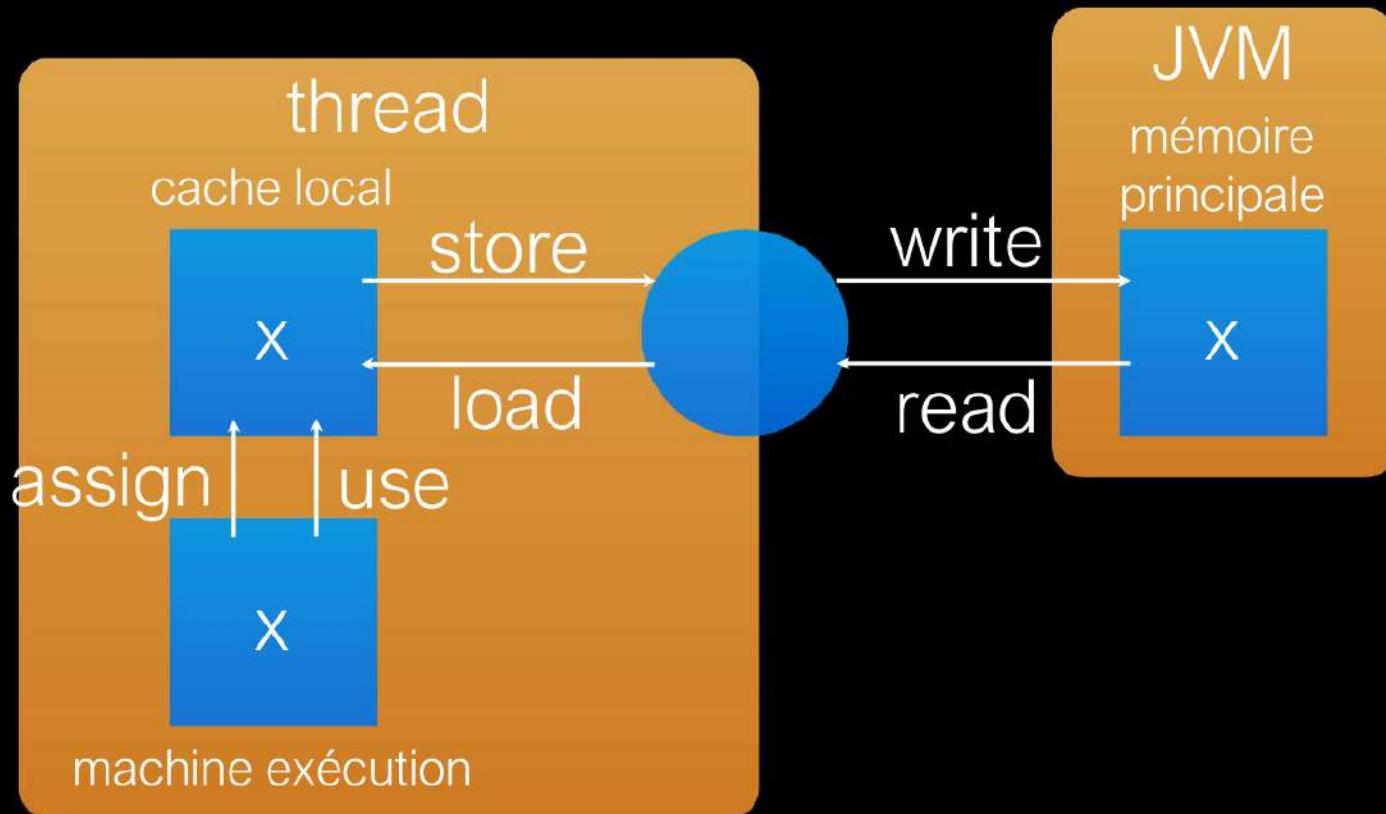
load v

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.ms<sub>1</sub>, alors **o** est verrouillé  
(t a le verrou de o)

puis :

- **t** a l'autorisation d'accéder à o.ms<sub>2</sub>, o.ms<sub>3</sub>, ...
- si **t'** appelle o.ms<sub>2</sub>, alors **t'** est bloqué  
(jusqu'à ce que le verrou sur o soit levé)
- **t'** est libre d'appeler o.m<sub>1</sub>, o.m<sub>2</sub>, ...

# 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);  
        }  
        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 fauchés!"); }  
  
    public void run() {  
        for (int i=1;i<10;i++) {  
            Liquide(100*i);  
            Thread.currentThread().sleep(50)  
        }  
    }  
}
```

# Actions atomiques et JVM

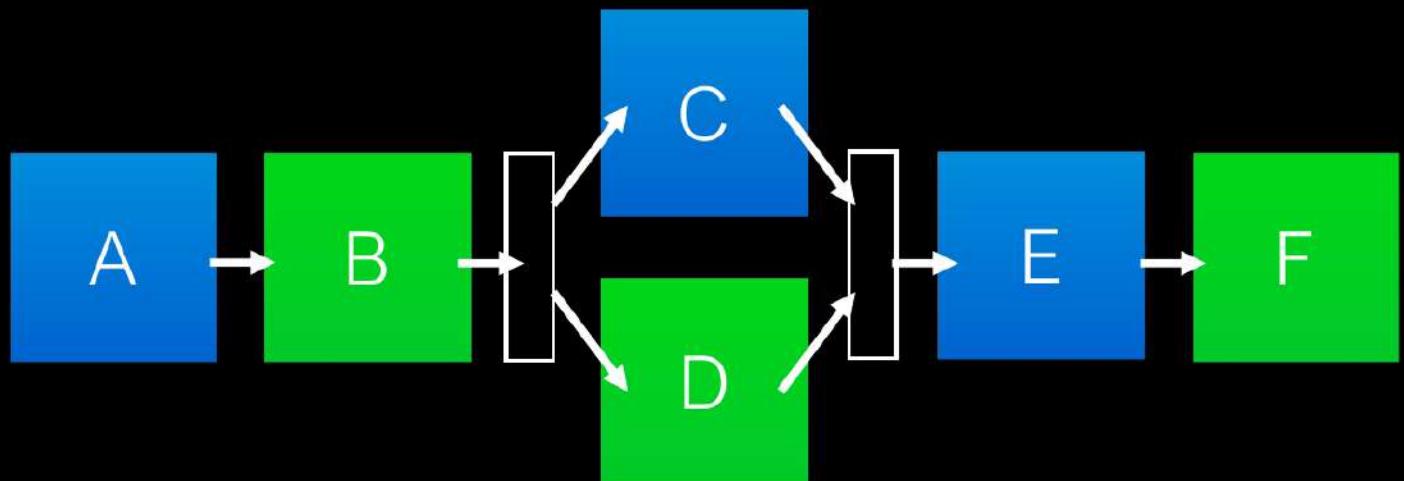
- pour un thread, on rajoute
  - lock (P)  
réclame à la mémoire p<sup>ale</sup> le verrou associé à un objet
  - unlock (V)  
libère le verrou associé à un objet
- appel d'une méthode synchronized o.m par t
  - si o non verrouillé, il le devient
  - si o verrouillé, t est bloqué (jusqu'à libération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

# 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<sub>1</sub> fait {A,C,E}
  - P<sub>2</sub> fait {B,D,F}
- on veut le workflow ci-dessus  
(en séquence ABCDEF ou ABDCEF)
- quels sont les comportements de P<sub>1</sub> et P<sub>2</sub> ?



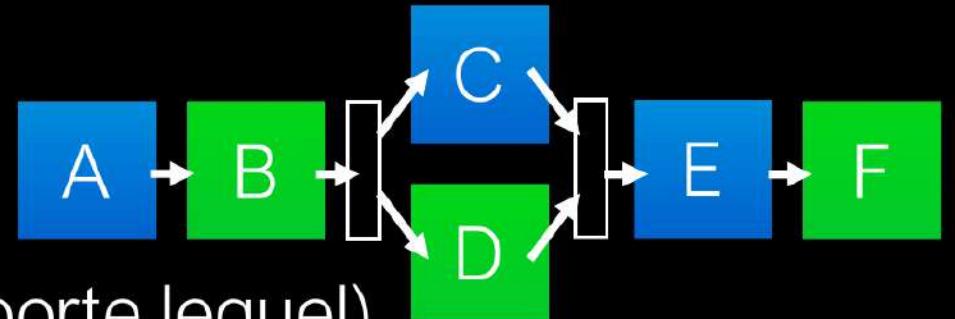
# wait et notify : principes (1/2)

- 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)

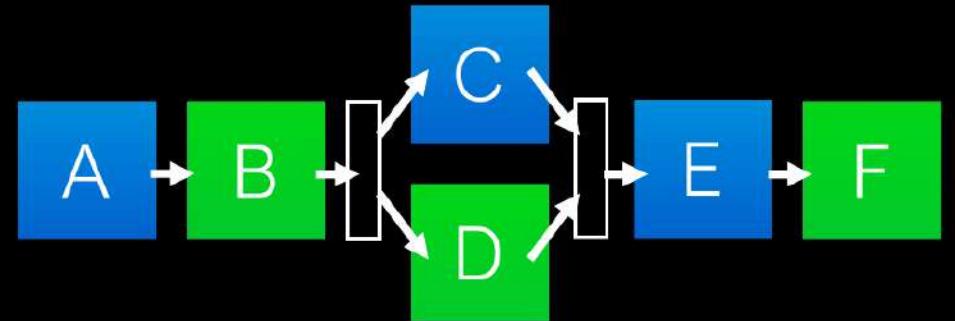
- attention !  
un thread qui appelle `wait` n'a pas moyen de s'auto-réveiller  
si tous les threads appellent `wait` sans qu'aucun n'appelle  
`notify` alors on aura un **blocage** (deadlock)
- attention ! (bis)  
les threads en attente ne sont pas réactivés si aucun thread  
ne travaille sur l'objet
- 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

# Solution des 2 processus ?



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

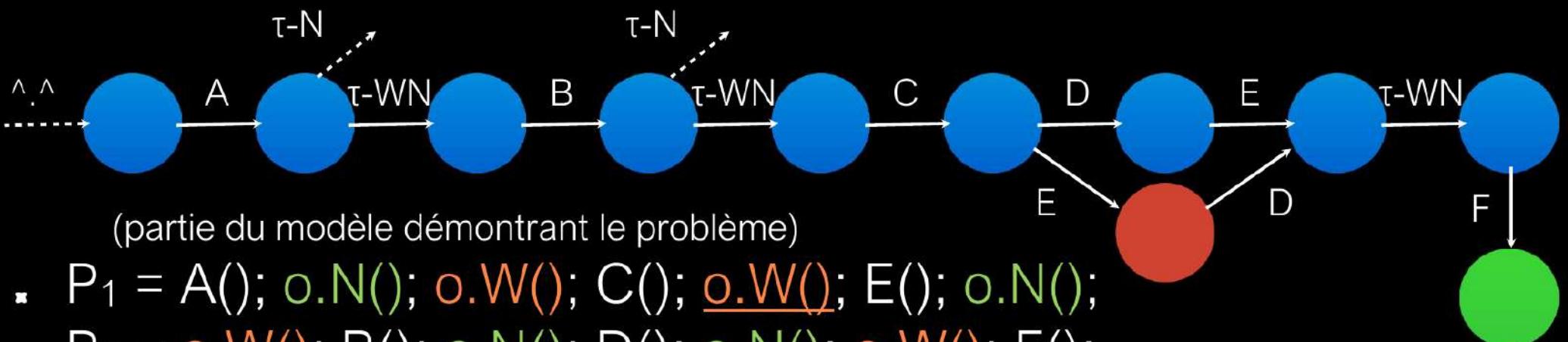
# Solution des 2 processus ?



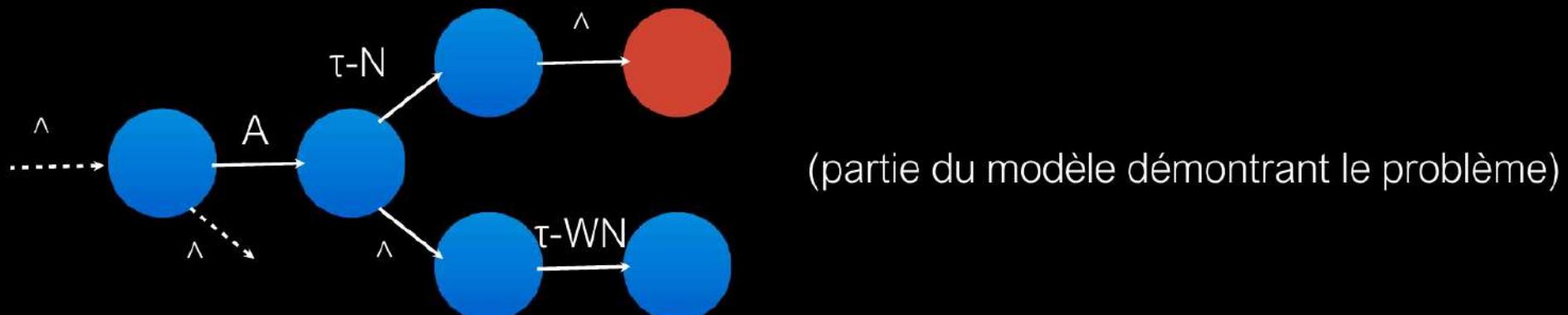
- cette solution est-elle correcte ?
  - $P_1 = A(); \text{ o.N(); o.W(); C(); E(); o.N();}$   
 $P_2 = \text{o.W(); B(); o.N(); D(); o.W(); F();}$
  - non : on peut avoir ABCEDF (exercice)
  - et celle-ci ?
    - $P_1 = A(); \text{ o.N(); o.W(); C(); } \underline{\text{o.W();}} \text{ E(); o.N();}$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

# Solution des 2 processus ?

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



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



# Solution des 2 processus ?

$P_1 = A(); o.N(); o.W(); C(); \underline{o.W()}; E(); o.N();$   
 $P_2 = o.W(); B(); o.N(); D(); \underline{o.N()}; o.W(); F();$

```
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();  
}
```

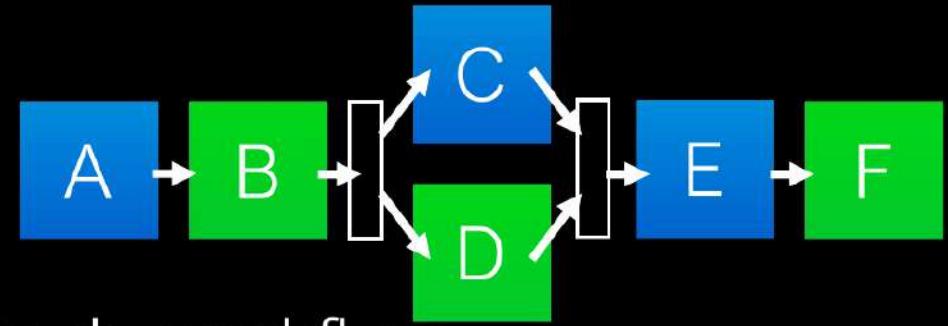
```
class P1 extends Thread {  
    Controller c;  
    public P1(Controller cx) { c=cx; }  
    public void A(...) { ... }  
    ... // definitions de C et E  
    //  
    public void run() {  
        yield();  
        A(...); c.notify();  
        c.wait(); C(...);  
        c.wait(); E(...); c.notify();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

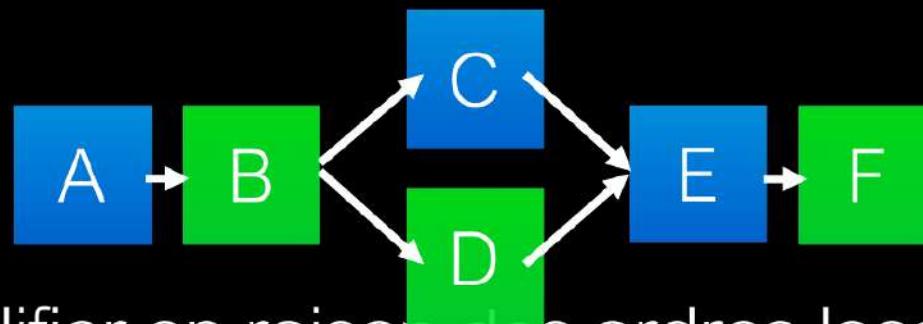
- principe :
  - section critique (SC)  
besoin d'exclusion mutuelle
  - un jeton  
celui qui l'a peut entrer en SC  
les autres bloquent
- implantation de haut niveau de P(.) et V(.) sur une variable partagée, ici le jeton (ou sémaphore)

```
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();  
    }  
}
```

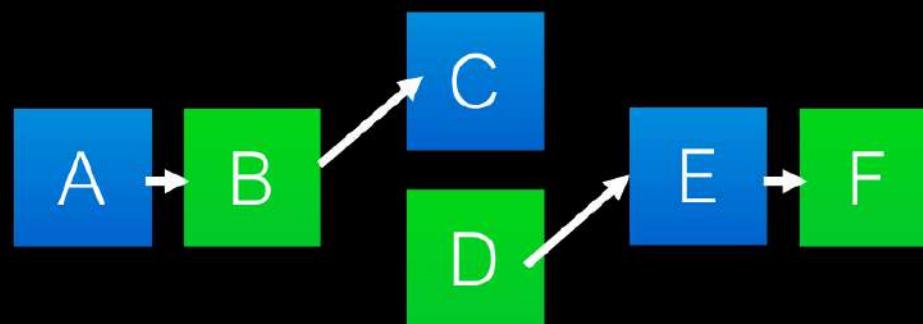
# Solution des 2 processus (1/3)



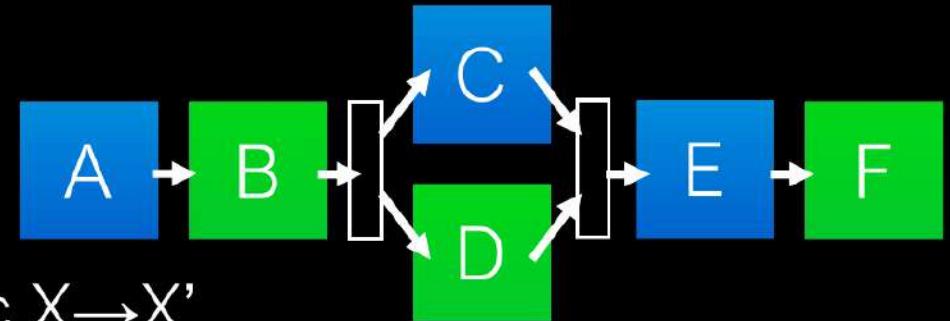
- définissons la structure causale du workflow



- on peut simplifier en raison des ordres locaux



# Solution des 2 processus (2/3)

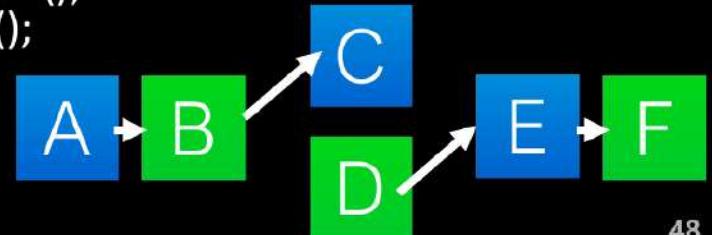


- pour chaque  $X$  tq il existe  $X'$  avec  $X \rightarrow X'$ ,  
on définit un couple de méthodes de contrôle :
  - `doneX() { isDoneX := true ; notify(); }`
  - `waitX() {if (! isDoneX) { try { wait(); } catch (...) {} }}`
- pour chaque processus  $P$ , pour chaque action  $Y$  de  $P$  tq  $X \rightarrow Y$  ( $X$  action de  $P'$ ) : ... `c.waitX(); Y(); ...`
- pour chaque processus  $P$ , pour chaque action  $X$  de  $P$  tq  $X \rightarrow Y$  ( $Y$  action de  $P'$ ) : ... `X(); c.doneX(); ...`
- $c$  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();  
        p1.join(); p2.join();  
    }  
}
```

```
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  $\{X_1, \dots, X_n\} \rightarrow Y : \dots \underline{c.waitX_1();} \dots; \underline{c.waitX_n();} Y(); \dots$