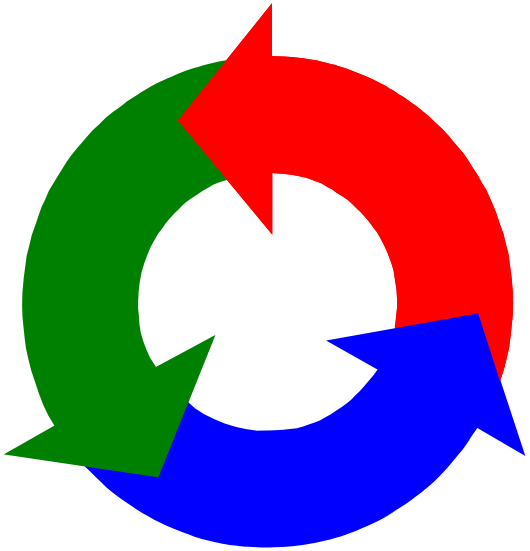
Répartition intra-applications :

Programmation parallèle: Le multi-Threading



Plan du cours

- Introduction
  - Définitions
  - Problématiques
  - Architectures de distribution
- **Distribution intra-applications**
  - Notion de processus
  - Programmation multi-thread
- Distribution inter-applications et inter-machines
  - Sockets
  - Middlewares par appel de procédures distantes
  - Middlewares par objets distribués (Java RMI, CORBA)
- Conclusion

Rappel de la définition d'un système réparti

- *Un système distribué est une collection de processus ou d'ordinateurs indépendants et coopératifs qui apparaissent à l'utilisateur comme un seul et unique système cohérent*

- ***Cas 1: Plusieurs processus ou programmes sur une même machine***
  - *On parle alors de parallélisme*
- ***Cas 2: Plusieurs processus ou programmes sur un ensemble de machines c à d:***
  - *Pas d'horloge commune => pas de datation possible des évènements*
  - *Plusieurs mémoires => pas de contrôle d'état*
  - *=> On parle alors de distribution*

Pourquoi on s'intéresse au cas du parallélisme?

- **Première forme de répartition.**
- **C'est un contexte simple pour évoquer**
  - La concurrence
  - La synchronisation
  - La notion d'état cohérent
  - Et
  - On peut simuler la répartition => utilisation des « *Pipes* ».
- **Surtout c'est pour montrer qu'il ne faut pas**
  - Compter sur les propriétés physiques d'une machine pour faire des applications distribuées et garantir certaines propriétés.
  - Notion de solution canonique.
- **Introduire les concepts et techniques pour faire face à la synchronisation.**
- **Un début pour la modélisation.**

Différence entre la programmation séquentielle et concurrente ?

- **Un programme séquentiel est destiné à être exécuté sur un même processeur (parfois c'est faux)**
- **Un programme parallèle peut être exécuté sur autant de processeur que de composantes parallèles du programme**
- **Notre façon de caractériser ses deux types de programme:**
  - Séquentiel : paire d'observable
    - entrées et sorties possibles associées.
    - Équivalence P1 et P2 si ils ont les mêmes observables
    - Si on insert P1 ou bien P2 dans un troisième P3 cela ne change pas les observables de P3
    - => programmes séquentiels pouvant être vues comme atomiques
  - Parallèle : P=P1 || P2
    - La coopération entre P1 et P2
    - => influencer leurs exécutions mutuelles
    - Ce qui rend les états intermédiaires de P importants pour sa caractérisation
    - Surtout les moments où P1 et P2 s'influencent mutuellement

Parallélisme et coopération

- **Parallélisme :**
  - Une collection de processus s'exécutant de manière concurrente (forte ou lâche).
  - Chaque processus est un programme séquentiel qui s'exécute pour l'objectif d'un programme golobal
- **Coopération**
  - L'application a un objectif global
  - L'objectif global de l'application est un ensemble de propriétés qui font intervenir :
    - L'état d'objets faisant partie des observables des processus.
    - Ou des relations entre les objets.
    - Ce qui pousse les processus à coopérer
    - =>Avancer de telle façon que l'objectif soit atteint.
  - Coopération se fait donc par :
    - Partage de la mémoire (l'objet de ce cours)
    - Ou inter-connecter par un moyen de communication (l'objet des cours suivants)

Parallélisme et coopération : synchronisation

- **Partage de mémoire**
  - P1 et P2 ont accès à la même mémoire (donc aux mêmes objets).
  - P1 influence P2 en changeant les valeurs des variables dont le comportement de P2 dépend.
  - Pour cela il faut que cette influence soit possible et contrôlable.
    - Problème: Les données peuvent être consultées pendant qu'elles sont en cours de changement.
    - Solution : **synchroniser l'utilisation des objets partagés**
- **Connexion aux réseaux de communication**
  - Chaque processus possède sa propre mémoire
  - Partage des canaux de communication où il peuvent s'envoyer des messages.
  - Les messages contiennent des données qui influencent le comportement des processus

Communication et synchronisation

- **Pour interagir les processus doivent communiquer**
- **Pour coopérer les processus doivent communiquer et synchroniser.**
- **Communication :**
  - Indirecte par changement de valeurs d'objet partagés
  - Directe par envoi de messages.
- **Synchronisation:**
  - Pour communiquer le processus change l'état d'un objet consulté par un autre.
  - Si l'autre consulte avant que l'état change ça peut donner un comportement indésirable ou des résultats indésirables
  - => communication doit être organisée entre les actions des processus : Synchronisation.

Communication et synchronisation (2)

- **Trois formes de synchronisation**
  - Exclusion mutuelle => grouper les événements ou encore actions dans des sections critiques.
    - Ses actions ne se chevauchent pas durant l'exécution.
    - Durant la section critique aucune variable (qui influence le comportement du programme) n'est changé
  - Synchronisation conditionnel => bloquer l'avancement d'un programme tant qu'une condition sur les données n'est atteinte
    - Il peut être utilisé pour implémenter la première forme
    - Exemple : Éviter qu'une valeur d'une variable soit lue avant qu'une modification de celle-ci ne soit terminée.
  - Synchronisation concerne l'acte de communication lui-même
    - L'acte d'envoi et de réception sur le canal est synchronisé sur l'envoi-réception de message
    - => communication synchrone
- **Les autres formes de communication sont asynchrones.**

Communication et synchronisation (3)

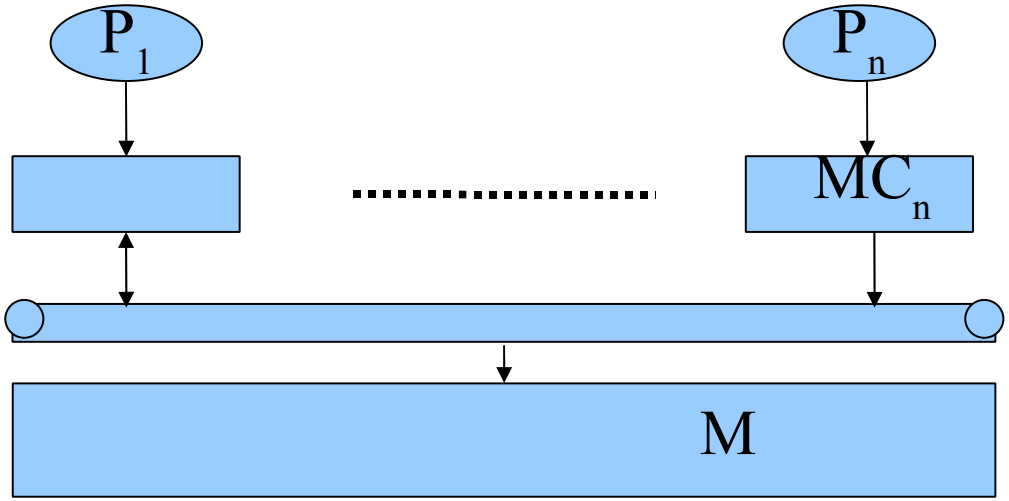
- **Exemple qui illustre les trois types de synchronisation c'est la communication entre deux processus.**
- **Les deux processus communiquent par dépôt et consommation de messages dans une zone tampon.**
  - Exclusion mutuelle doit être utilisée pour prévenir que la lecture et écriture ne se chevauchent (pas de messages partiellement lus ni partiellement écrits) => lecture de message dans son intégralité
  - Synchronisation conditionnelle
    - L'émetteur ne doit pas écrire deux fois de suite => écrit ssi le message a été consommé
    - Le récepteur ne doit pas lire le message deux fois => ne lit que ssi un nouveau message a été déposé.
    - Une communication synchrone utilise un tel mécanisme à des niveau bas pour donner l'impression d'une communication instantanée.

Mais avant : voyons les états des machines dites parallèles et les systèmes d'exploitation

- **Une machine parallèle est essentiellement un ensemble de processeurs qui coopèrent et communiquent.**
- **On distingue classiquement quatre types de parallélisme. Cette classification est basée sur les notions:**
  - Flot de contrôle (deux premières lettres, I voulant dire ``Instruction'')
  - Flot de données (deux dernières lettres, D voulant dire ``Data'').
- **SISD (*Single Instruction Single Data*)**
  - Cas de machine séquentielle (von neuman)
- **SIMD**
  - Exécution en parallèle de la même instruction se fait en même temps sur des processeurs différents (parallélisme de données synchrones)
  - Exemple : Sommer les lignes d'une matrice
- **MISD**
  - Plusieurs instruction (différentes) en même temps sur la même donnée.
  - Cas de calcul vectoriel
- **MIMD.**
  - Cas qui nous intéresse dans ce cours.

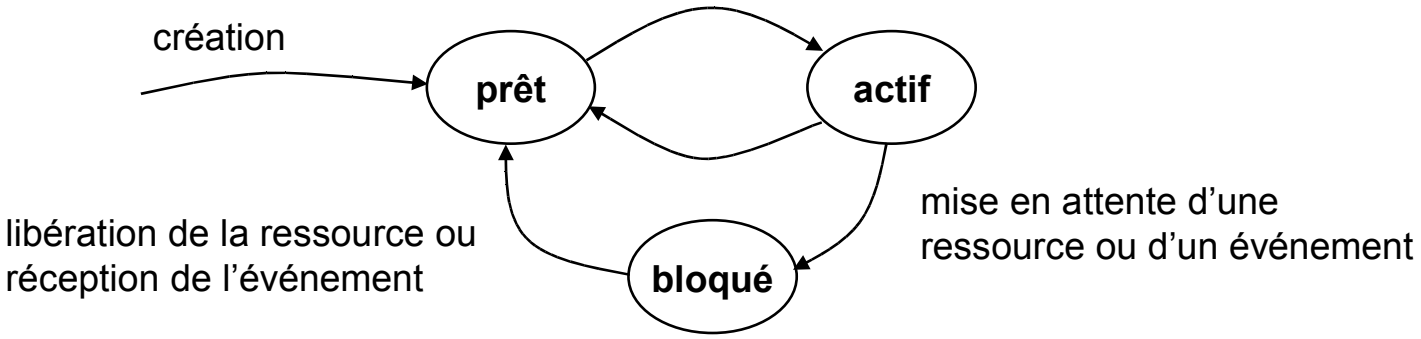
MIMD

- **Représente les deux cas qui nous intéréssent:**
  - Mémoire partagée
  - Mémoire distribuée + communication.
- **On va illustrer par le langage java :**



Systèmes d'exploitation multitâches

- **Principe**
  - Exécution « simultanée » de plusieurs programmes ou **processus** ou **tâches**
  - un seul programme peut s'exécuter à un instant donné (si un seul CPU)
  - nécessité de partager le temps CPU = **ordonnancement**
    - le système d'exploitation découpe le temps en tranches (quanta)
    - il attribue chaque quantum de temps à une tâche (système en temps partagé)
    - différentes politiques d'ordonnancement possibles
  - pseudo-parallélisme
- **Principaux états d'un processus**



Préemptif et coopératif

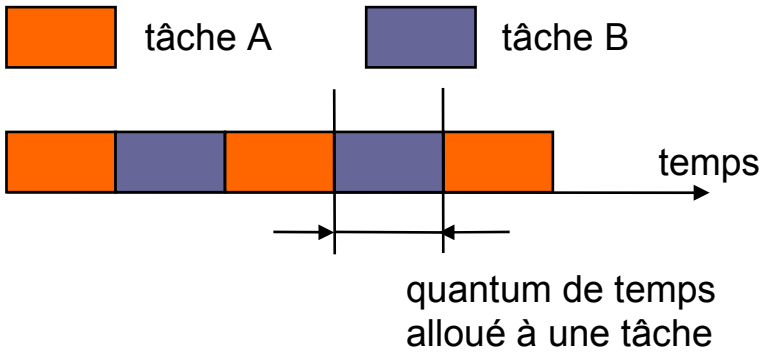
- **2 sortes de multitâche**
  - multitâche coopératif ou ***non-préemptif*** = chaque programme ne peut être interrompu que s'il en fournit explicitement l'autorisation
    - ex. : Windows 3.1
    - pb = un programme mal conçu peut bloquer le système indéfiniment
  - multitâche ***préemptif*** = interrompt les tâches sans les consulter
    - ex. : Unix, Linux, Windows NT
    + plus efficace
    − plus difficile à implémenter
    − plus difficile de contrôler finement l'exécution de son programme

Politiques d'ordonnancement

- **Répartition manuelle**
  - prévoir l'ensemble des contraintes temporelles
  - définir un découpage des tâches
  - définir une alternance des exécutions des phases qui respectent les contraintes du système de tâches
- **Répartition automatique**
  - répartition naïve « équitable »
  - répartition intelligente (priorités, échéancier, etc.)
  - différentes approches
    - en fonction d'un événement
      - ex. : les systèmes « pulled-loop »
    - en fonction d'un état
      - ex. : les systèmes « state-driven »
    - en fonction d'une répartition du temps
      - les systèmes de « coroutines »
      - les systèmes « interrupt-driven »

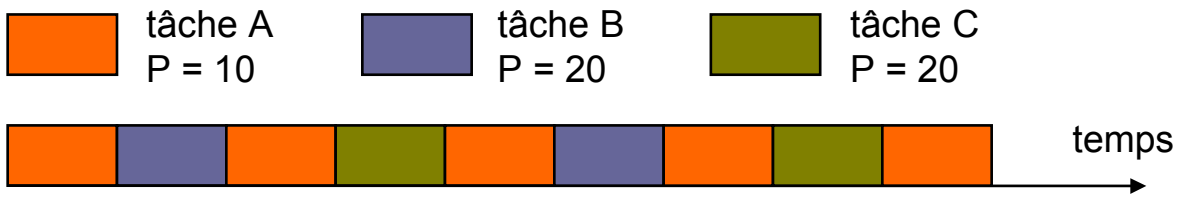
Ordonnancement « interrupt-driven » (1)

- **Round Robin**
  - le temps est équitablement distribué entre les différentes tâches



Ordonnancement « interrupt-driven » (2)

- **Preemptive Priority System**
  - priorité associée à chaque tâche
  - le processeur est partagé entre les tâches de plus haute priorité
  - mécanismes de suspension pour permettre aux tâches de plus faible priorité de s'exécuter
- **Rate Monotonic**
  - la fréquence d'exécution de chaque tâche est proportionnelle à sa priorité



Les processus

- **Un processus est défini par**
  - son bloc de contrôle
    - priorité
    - point d'entrée
    - contexte (pile, registres)
    - état
  - son code
  - sa périodicité
- **un système multi-tâche comprend**
  - un ensemble de tâches utilisateur
  - une tâche oisive
    - tâche de + faible priorité
    - ne s'exécute que si toutes les autres sont suspendues ou détruites
    - ne peut être suspendue
    - sert à occuper le processeur
  - un gestionnaire d'interruptions

Conclusion sur les matériels et Systèmes d'exploitation

- **Plusieurs formes de machine parralléle.**
- **Divers approches pour le fonctionnement des SE(s).**
- **Systèmes d'exploitation peuvent héberger plusieurs applications :**
  - Vue d'une application : elle s'exécute dans un environnement incertain
  - => difficulté d'avoir un contrôle total sur les mécanismes offerts par les systèmes.
- **Solution :**
  - Passage à un niveau plus haut : les langages de programmation
  - Système canonique pour le contrôle du parallélisme.
  - Mécanisme similaire à ceux utilisés aux niveaux plus bas.
- **Conseil : il faut toujours raisonner au pire des cas et contrôler soi-même l'exécution de ses programmes.**

Dans ce cours on va illustrer avec les processus de java (Multi-Threading)

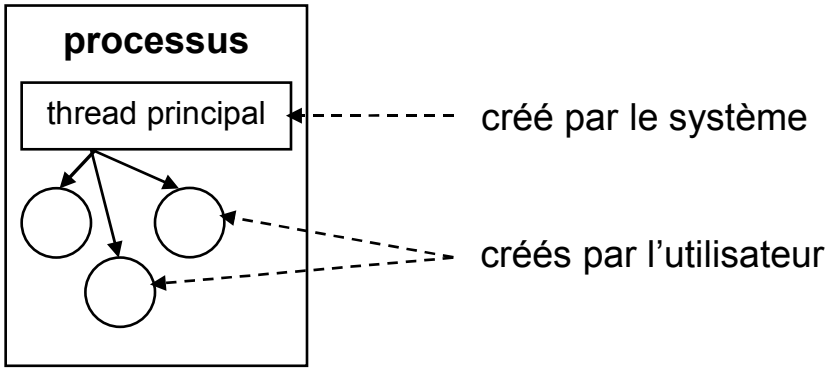
- **Une application java peut contenir plusieurs processus.**
- **Les processus en java sont appélés « thread » : processus légers.**
- **Il existe toujours un premier thread qui commence à exécuter la fonction *main*() (*JVM*).**
  - Correspond généralement à un processus lourd (SE)
- **Dans les interfaces graphiques**
  - Il existe un thread qui attend les actions de l'utilisateur et déclenche les écouteurs
  - Il existe un autre thread qui redessine les composants graphiques qui ont besoin de l'être.
- **On peut créer ses propres threads**

Comment créer un thread?

- **Avant tout un thread est une Classe**
- **Cette classe contient une méthode spécifique *run()*.**
- **Une fois appelé, un processus est crée et se détache du processus appelant. Ils deviennent parallèles.**
- **Une solution consiste à hériter de Thread et surcharger la méthode *run()*.**
- **Très contraignant comme approche car pas d'héritage multiple en java.**
- **Solution :**
  - passer par l'interface ***Runnable*** (utilisée par Thread) pour implementer ***run()***
- **Attacher votre *Runnable* à un objet Thread et exécutez votre processus.**

Applications multithreads

- **Principe des threads**
  - idée = appliquer le principe du multitâche au niveau des applications
    - application effectue plusieurs tâches en même temps
    - chaque tâche est appelée un thread
    - application **multithread** = peut exécuter plusieurs threads à la fois
  - thread = processus léger
  - les threads se partagent la mémoire du processus
- **Exemples**
  - ramasse-miettes Java
  - interfaces graphiques
  - navigateur Web



Threads vs. processus

- **Gestion de la mémoire**
  - un processus a une copie unique de ses propres variables
  - les threads d'une application partagent les mêmes données
    - risque de pbs d'accès concurrents
- **Efficacité**
  - bien plus rapide de créer/détruire des threads que des processus
  - communication entre processus plus lente et plus restrictive

Exemple – une balle qui rebondit version 1.0

- **Animation**
  - une balle qui se déplace en ligne droite
  - quand elle arrive sur un bord, elle rebondit

Bounce – Sans Thread (1)

```
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);
    }
  }
}
```

Bounce – Sans Thread (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, XSIZE,
    YSIZE));
  }

  public void go() {
    try {
      while (true) {
        if (!freeze)
       move();
        Thread.sleep(5);
      }
    }
    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());
  }
}
```

Bounce – Limites de la solution sans Thread

- problème = l'animation de la balle bloque l'application
  - les boutons ne réagissent plus
  - impossible de fermer la fenêtre
- solution = séparer l'animation du reste de l'interface graphique
  - le thread principal gère l'interface graphique
  - un nouveau thread est créé pour gérer l'animation de la balle

Exemple – version 2.0 solution avec Thread

- **Transformer la classe Ball en thread**
  - étendre la classe Thread
  - surcharger la méthode run()

    ```
    public void run()
    ```
- **Créer et démarrer un thread pour chaque balle**
  - créer une instance de la classe Ball

    ```
    Ball b = new Ball(...);
    ```

    le thread ne fonctionne pas encore
  - démarrer le thread en appelant la méthode start()

    ```
    b.start();
    ```
    - la machine virtuelle démarre le Thread en appelant la méthode run() quand il est prêt à être exécuté
    - ne pas appeler la méthode run directement => la méthode run serait exécutée dans le même thread

BounceThread – code modifié (1)

```
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 boolean freeze = false;

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

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

  public void go() {
    try {
      while (true) {
        if (!freeze)
       move();
         Thread.sleep(5);
      }
    }
    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 boolean freeze = false;

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

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

  public void run() {
    try {
      while (true) {
        if (!freeze)
       move();
         sleep(5);
      }
    }
    catch (InterruptedException exception) {}
  }

  ...
}
```

BounceThread – code modifié (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);
    }
  }
}
```

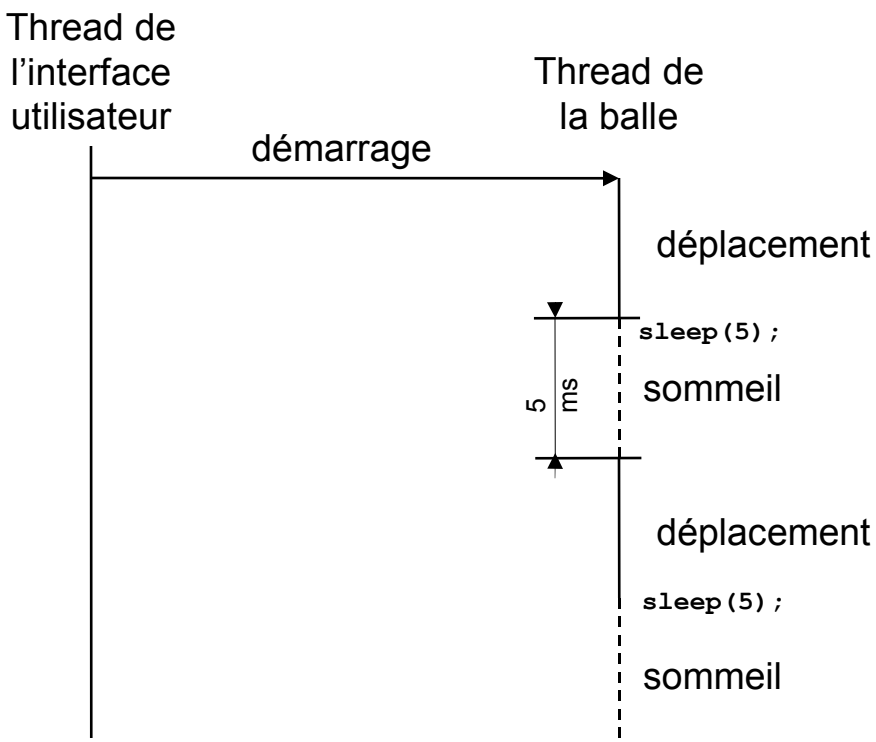


```
class BallCanvas extends JPanel {
  private ArrayList balls = new ArrayList();

  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);
    }
  }
}
```

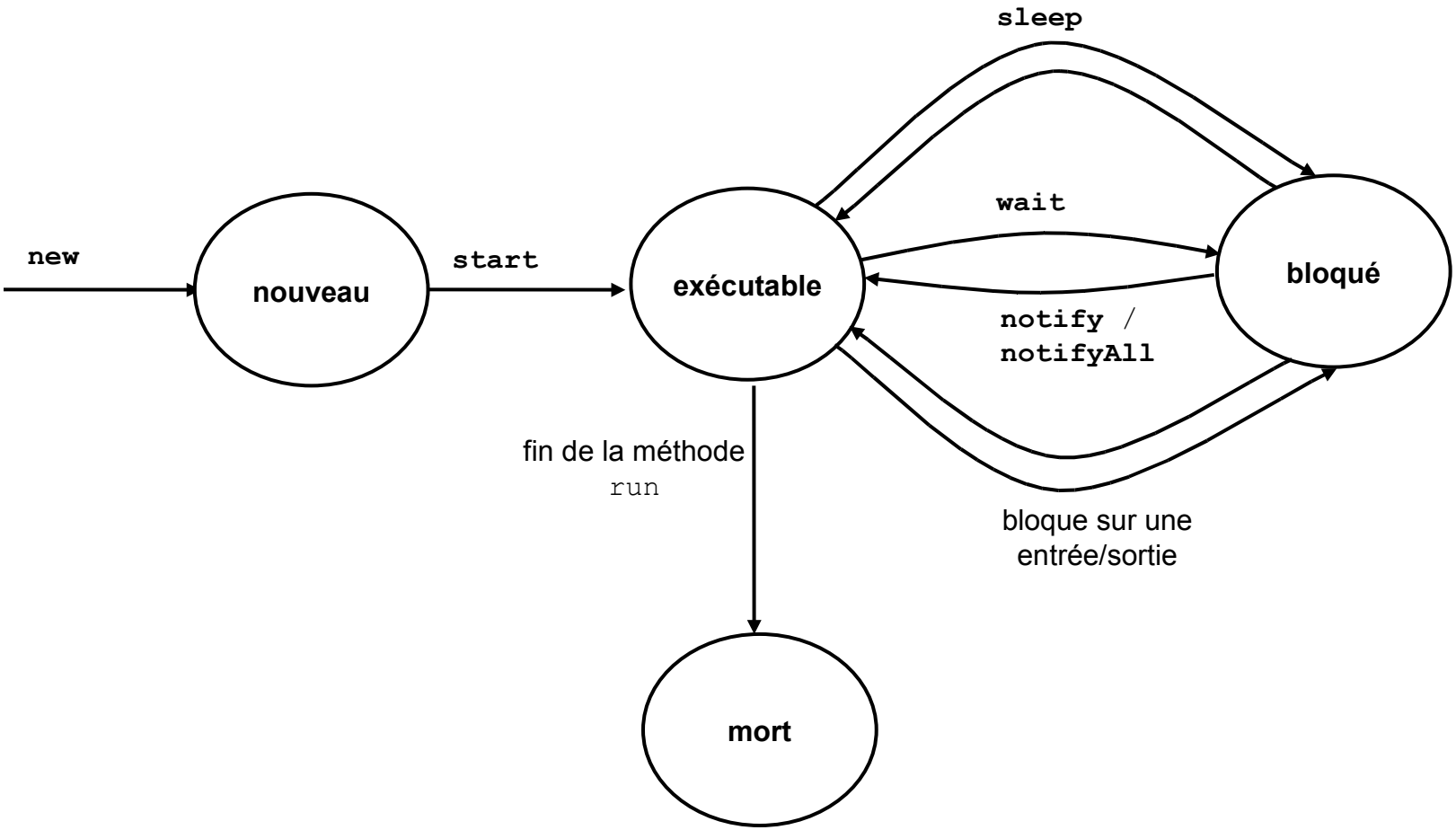
Java.lang.Thread

- **`Thread()`**
  - construit un nouveau thread
- **`void run()`**
  - méthode principale exécutée par le thread
  - doit être surchargée
- **`void start()`**
  - lance le thread et appelle sa méthode **`run`**
  - revient immédiatement
  - le nouveau thread est exécuté en parallèle
- **`static void sleep(long ms)`**
  - place le thread en état de veille pendant le nombre de millisecondes spécifié

Exemple – une balle qui rebondit version 2.0



Etats d'un threads (1)



Etats d'un threads (2)

- **Nouveau**
  - suite à la création du thread par **`new`**
  - pas encore exécutable directement. Manquent :
    - certaines initialisations
    - l'allocation des ressources nécessaires à l'exécution du thread
- **Exécutable**
  - suite au démarrage du thread par **`start`**
  - « exécutable » ≠ « en cours d'exécution »
    - la doc Java ne les distingue pas comme des états séparés
    - c'est l'OS qui gère le passage de l'un à l'autre
    - gestion ≠ suivant les OS
      - Solaris => PPS
      - Windows NT => Round Robin

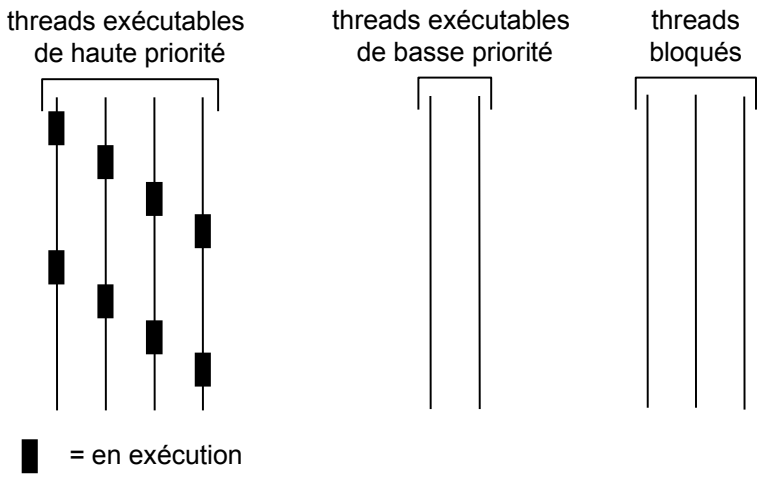
Etats d'un threads (3)

- **Bloqué**
  - suite à l'une des actions suivantes
    - appel de la méthode **`sleep()`**
      - retour à la fin du temps spécifié (en millisecondes)
    - opération bloquante sur des E/S
      - retour quand l'opération est terminée
    - appel de la méthode **`wait()`**
      - retour quand un autre thread appelle **`notify()`** ou **`notifyAll()`**
    - tentative pour verrouiller un objet déjà verrouillé par un autre thread
      - retour quand l'autre thread libère le verrou
    - appel de la méthode **`suspend()`** => à éviter
      - retour après appel de sa méthode **`resume()`**
- **Mort**
  - fin normale de la méthode **`run()`**
  - exception non récupérée
  - Stop => à éviter (*deprecated*)

Java.lang.Thread

- **`boolean isAlive()`**
  - renvoie **`true`** si le thread est démarré et n'est pas encore terminé
- **`void suspend() // deprecated`**
  - suspend l'exécution du thread
- **`void resume() // deprecated`**
  - reprend l'exécution du thread (uniquement valide si **`suspend`** a été invoquée précédemment)
- **`void stop() // deprecated`**
  - interrompt le thread
- **`void join()`**
  - attend que le thread ait cessé d'être vivant
- **`void join(long ms)`**
  - attend que le thread ait cessé d'être vivant ou que le nb de millisecondes spécifié se soit écoulé

Exemple – plusieurs balles qui rebondissent



Interrompre des threads (1)

- **Principe**
  - un thread s'arrête quand sa méthode **`run()`** est terminée
    - pas de méthode intégrée pour mettre fin à un thread
    - la méthode **`run()`** doit vérifier régulièrement si elle doit terminer

```
public void run() {
  ...
  while (pas de requête de fin && encore du travail) {
    faire le travail
    se mettre en sommeil
  }
  // sort de la méthode run et met fin au travail
}
```

Interrompre des threads (2)

- Pb = un thread bien conçu (qui se met en sommeil régulièrement) ne peut pas déterminer s'il doit s'arrêter pendant qu'il dort
- Solution = utiliser la méthode **`interrupt()`**
  - l'appel de la méthode **`interrupt()`** sur un thread bloqué provoque l'arrêt de l'appel bloquant (**`sleep`** ou **`wait`**) par une **`InterruptedException`**
  - le thread qui intercepte une **`InterruptedException`** peut choisir la manière dont il réagit

```
public void run() {
  try {
    ...
    while (encore du travail) {
      faire le travail
      se mettre en sommeil
    }
  }
  catch (InterruptedException e) {
    // le thread a été interrompu pendant sleep ou wait
  }
  // sort de la méthode run et met fin au thread
}
```

Interrompre des threads (3)

- Pb = pas d'exception levée si l'appel à **`interrupt()`** se produit
  - pendant que le thread est actif
  - pendant que le thread est bloqué sur une opération d'entrée/sortie
- Solution = utiliser la méthode **`interrupted()`**
  - appeler **`interrupted()`** pour savoir si le thread a été récemment interrompu

```
public void run() {
  try {
    ...
    while (!interrupted() && encore du travail) {
      faire le travail
      se mettre en sommeil
    }
  }
  catch (InterruptedException e) {
    // le thread a été interrompu pendant sleep ou wait
  }
  // sort de la méthode run et met fin au thread
}
```

BounceThreadInterrupted – code modifié (1)

```
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);
    }
  }
}
```

BounceThreadInterrupted – code modifié (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 boolean freeze = false;

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

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

  public void run() {
    try {
      while (true) {
        if (!freeze)
       move();
         sleep(5);
      }
    }
    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 boolean freeze = false;

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

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

  public void run() {
    try {
      while (!interrupted()) {
        if (!freeze)
       move();
         sleep(5);
      }
    }
    catch (InterruptedException exception) {}
  }

  ...
}
```

Java.lang.Thread

- **`void interrupt()`**
  - envoie une demande d'interruption à ce thread
  - le flag **`interrompu`** du thread est mis à **`true`**
  - si le thread interrompu est bloqué par un appel à **`sleep`** ou **`wait`**, une **`interruptedException`** est levée
- **`static boolean interrupted()`**
  - examine si le thread courant a été interrompu
  - positionne le flag **`interrompu`** à false
- **`boolean isInterrupted()`**
  - examine si un thread a été interrompu
  - ne modifie pas le flag **`interrompu`**

Priorités d'un thread

- **Chaque thread possède une priorité**
  - par défaut, la priorité du thread parent
  - comprise entre **`MIN_PRIORITY`** (1) et **`MAX_PRIORITY`** (10)
  - **`NORM_PRIORITY`** vaut 5
  - priorité modifiable avec la méthode **`setPriority`**
- **Principe**
  - le gestionnaire choisit *généralement* le thread de *plus haute priorité actuellement exécutable*
  - le thread continue son exécution jusqu'à ce que
    - il s'arrête en appelant la méthode **`yield`**
    - il cesse d'être exécutable (en mourrant ou en étant bloqué)
    - un thread de priorité supérieure devienne exécutable
  - le gestionnaire sélectionne un nouveau thread, choisi parmi ceux qui sont exécutables et ont la plus haute priorité

BounceExpress – code modifié

```
class BounceFrame extends JFrame {
  public BounceFrame() {
    ...
  }
}
```

```
class Ball extends Thread {
  ...

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

  ...
}
```



```
class BounceFrame extends JFrame {
  public BounceFrame() {
    ...
    addButton(buttonPanel, "Course", new
    ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        for (int i = 0; i < 10; i++)
          addBall(Thread.NORM_PRIORITY + 2, Color.red);
        for (int i = 0; i < 10; i++)
          addBall(Thread.NORM_PRIORITY, Color.black);
      }});

    ...
  }
}

class Ball extends Thread {
  ...

  public Ball(Component c, int priority,
              Color aColor) {
    canvas = c;
    setPriority(priority);
    color = aColor;
  }

  ...
}
```

Priorités d'un thread (2)

- **Problèmes**
  - le gestionnaire de threads est à la merci de l'implantation de la gestion des threads sur la plate-forme hôte
    - toutes les plates-formes n'ont pas 10 niveaux de priorité
    - comportement potentiellement différent selon la plate-forme
    - possibilité de traitement inéquitable entre les threads
    - possibilité de comportement différent suivant la charge de la machine
  - Mise au point délicate (voire impossible)

Java.lang.Thread

- **`void setPriority(int newPriority)`**
  - définit la priorité de ce thread
- **`static int MIN_PRIORITY`**
  - définit la priorité minimale qu'un thread peut avoir (1)
- **`static int NORM_PRIORITY`**
  - définit la priorité par défaut d'un thread (5)
- **`static int MAX_PRIORITY`**
  - définit la priorité maximale qu'un thread peut avoir (10)
- **`static void yield()`**
  - arrête le thread en cours d'exécution
  - donne la main aux autres thread exécutables de priorité au moins égale

Threads coopératifs et égoïstes

- **Donner aux autres une chance d'être exécutés**
  - sleep : se placer en sommeil pendant un temps déterminé
  - yield : autoriser le gestionnaire à l'interrompre
  - un thread qui n'appelle ni l'un ni l'autre pendant une longue boucle est dit « égoïste »
- **Comportement**
  - dépend du système et de la politique d'ordonnancement des threads
    - basée sur les priorités
    - basée sur un découpage du temps
  - il vaut mieux prévoir le pire des cas
    - appeler sleep ou yield dans chaque boucle

BounceSelfish – code modifié

```
class Ball extends Thread {
  ...

  public void run() {
    try {
     while (true) {
        if (!freeze)
      move();
        sleep(5);
      }
    }
    catch (InterruptedException exception) {}
  }

  ...
}
```



```
class Ball extends Thread {
  ...

  public void run() {
    try {
      while (true) {
        if (!freeze)
          move();
        if (selfish) {
          // attente active
          long t = System.currentTimeMillis();
          while (System.currentTimeMillis() < t+5)
            ;
        }
        else
          sleep(5);
      }
    }
    catch (InterruptedException exception) {}
  }

  ...
}
```

Groupes de threads

- **Principe**
  - dans certains cas, l'application repose sur de nombreux threads
  - utile de les regrouper par fonctionnalités
  - possible de les manipuler de manière groupée

```
ThreadGroup g = new ThreadGroup("balls");
Thread t = new Thread(g, "ball");
```

  - possible de définir des sous-groupes enfants

BounceThreadGroup – code modifié (1)

```
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);
    }
  }
}
```

→

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

  public void addBall() {
    Ball b = new Ball(this, ballsGroup);
    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 interruptAll() {
    ballsGroup.interrupt();
    balls = new ArrayList();
    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);
    }
  }
}
```

BounceThreadGroup – code modifié (2)

```
class Ball extends Thread {
  ...

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

  ...
}
```

```
class Ball extends Thread {
  ...

  public Ball(Component c, ThreadGroup group) {
    super(group, "ball");
    canvas = c;
  }

  ...
}
```

Java.lang.ThreadGroup

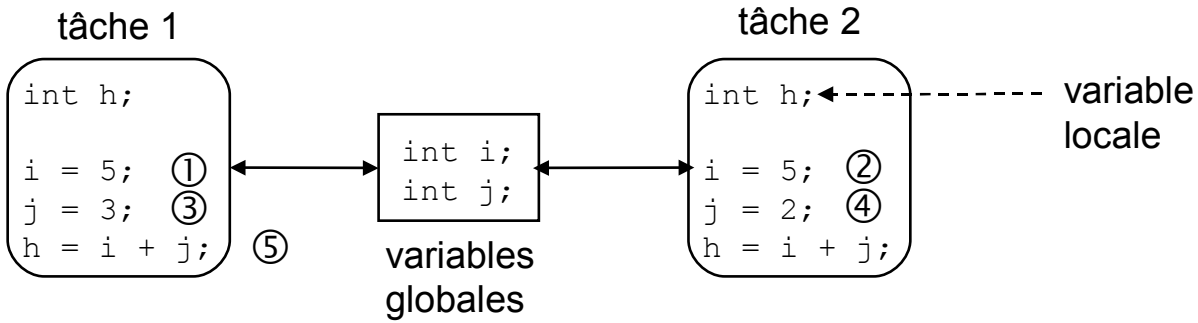
- **`ThreadGroup(String name)`**
  - crée un nouveau ThreadGroup
  - son parent est le groupe du thread courant
- **`ThreadGroup(ThreadGroup parent, String name)`**
  - crée un nouveau ThreadGroup
- **`int activeCount()`**
  - renvoie le nb de threads actifs dans le groupe
- **`int enumerate(Thread[] list)`**
  - renvoie les références de tous les threads actifs dans le groupe
- **`ThreadGroup getParent()`**
  - renvoie le parent de ce groupe de threads
- **`void interrupt()`**
  - interrompt tous les threads du groupe et de tous ses groupes enfants

Java.lang.Thread

- **`Thread(ThreadGroup g, String name)`**
  - crée un nouveau thread qui appartient au groupe spécifié
- **`ThreadGroup getThreadGroup()`**
  - renvoie le groupe de threads de ce thread

Problème de ré-entrance : test & set

- **Ré-entrance**
  - accès concurrent à une ressource
  - partage de données par une zone de mémoire commune à deux tâches



- **Test & set**
  - pb d'atomicité des opérations

```
if (solde - retrait > 0)
  solde = solde - retrait;
```

Exemple de Mr and Mrs Smith (1)

```
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) {
                        valeur -= somme;
            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)
       } }   }
```

Exemple de Mr and Mrs Smith (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();
}
```

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

Mrs Smith	Mr Smith
100	100
200	200
300	300
600	600

Explication

- **Simple si on sait deux choses:**
  - Quelles sont les actions atomiques de chaque processus.
  - Comment le système exécute les deux processus de Mr et Mrs
    - Mais avant comment il execute un programme séquentiel
- **Pour me croire il faut que la trace que je vous ai montrée soit possible sous les hypothèses de**
- **Pour cela on doit connaître la sémantique de java.**
- **Prenons un langage jouet :**
  - On suppose que la lecture et l'écriture sont atomiques.
  - Domaine des variables et constantes numériques

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

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

Pour comprendre ce qui se passe il faut un peu de sémantique(1)

- **Appelons *L* le langage engendré par cette grammaire. On définit la sémantique d'une façon inductive sur la syntaxe, en lui associant ce que l'on appelle un système de transitions.**
  - Représente un modèle pour état-comportement associé à chaque expression valide de *L*
- **Un système de transitions est un n-uplet (S, s_0,F,E,Tran) où,**
  - *S* est un ensemble d'états possibles du programme,
  - $s_0 \in S$ est l'état initial",
  - F $\subseteq S$ *: ensemble d'états finaux. (Acceptables)*
  - *E* est un ensemble d'étiquettes (événement atomique) on va
  - *Tran* $\subseteq S \times E \times S$ est l'ensemble des transitions. On représente plus communément une transition *(s,e,s')* ∈ Tran par le graphe: $s —^{e}—> s'$
- **Ce système est obtenu par application des règles de sémantique opérationnelle associées aux différents constructeurs du langage *L*.**

Construction du modèle

- **Comment on construit ce système de transition?**
  - Attribution d'une sémantique opérationnelle aux constructeurs du langage.
  - Le sens operationnel d'un constructeur est défini en fonction du comportement des parties qui le composent face aux actions possibles.
- **Notion d'état d'un programme**
  - Mémoire + comportement
    - Mémoire : c'est l'état des variables du programme
    - Comportement : ce qui reste du programme (des actions à faire)
  - Soit *N* l'ensemble de noms de variable qui prennent leurs valeurs dans *Z*
  - L'état d'un programme est le couple (l,H)
    - $l \in L$: état le code qui reste à évaluer.
    - $H \in \mathbb{Z}^{|V|}$ l'état des variables (avec V= Var(P) $\subseteq$N). environnement
- **Exprimer la sémantique consiste à exprimer l'évolution de chaque couple *(l,H)* face aux évènements du langage.**
- **Exemple:**
  - *(x=expr;block,H)* —$^{x=expr}$—> *(block,H'=[x—> [[expr]](H)])* *où [x—>[expr]] (H)* dénote l'affectation de la valeur entière résultat de l'évaluation de l'expression *expr* dans l'environnement *H* à *x*
  - *(if t then b1 else b2; b3,H)* —^{then}—> *(b1; b3,H) si [[t]] (H) = vrai sinon*
  - *(if t then b1 else b2; b3,H)* —^{else}—> *(b2; b3,H)*

une sémantique par entrelacement pour le parrallélisme

- **Soit le programme $P=(l_0,H_0)$**
- **Les traces d'éxecution possibles de P est**
  - Un système de transitions défini par le n-uplet $LTS^P=(S,s_0,F,E,Tran)$ où,
    - $S \subseteq (L \times Z^{|var(P)|})$ est un ensemble d'états possibles du programme,
    - $s_0=(l_0,H_O)$,
    - $F \subseteq S$: l'ensemble d'état finaux
    - E est un ensemble d'étiquettes (événements atomiques)
    - $Tran \subseteq S \times E \times S$ obtenue par application des règles sémantiques.
- **Soit maintenant un ensemble $\{P_1...P_i=(l_{i0},H_{i0})....P_n\}$**
  - On associ à chaque programme son $LTS^{Pi}=(S_i,s_{0i},F_i,E_i\ Trans_i)$
- **On introduit le programme P défini par la mise en parallèle des P_i Noté $P= P_1||...P_i...||P_n$**
- **Pour pouvoir raisonner sur P il faut voir la notion d'historique d'éxecution possible.**
  - La sémantique opérationnelle de cette composition parallèle
  - De manière compostionelle => le comportement de P est defini en fonction de celui des P_i

modélisation des variables partagées.

- **Les évènements d'un processus sont éxécutés séquentiellement selon la sémantique associée.**
- **Aucun contrôle sur le choix du processus. Tous les entrelacements sont possibles => sémantique par entrelacement**
- **À chaque P_j on associe par *[[.]]* définit plus haut un système de transitions $(S_j,i_j,F_j,E_j,Tran_j)$,**
- **On définit $(S,s_0,F,E,Tran)$=*[[P1* || ... || *Pn]]* par:**
  - $S=S_1 \times \ldots \times S_n$,
  - $s_0=(s_{01},\ldots,s_{0n})$,
  - $F=F_1 \times \ldots \times F_n$
  - $E=E_1 \cup \ldots \cup E_n$,
  - $Tran= \{ (s_1,\ldots,s_j,\ldots,s_n) \xrightarrow{tj} (s_1,\ldots,s_j',\ldots,s_n)$ où $s_j \xrightarrow{ti} s_j' \in Tran_j \}$
- **Mais on n' a pas encore fini:**
  - Supposons que $Var(P_1) \cap \ldots \cap Var(P_n) \neq \emptyset$ c à d des variables partagées. Or dans la sémantique chaque processus possède son propre environnement.
  - Soit $VP = Var(P_1) \cap \ldots \cap Var(P_n)$
  - Solution : on isole toutes les variables partagées dans un état globale noté ***Sg***

Sémantique du paraléllisme revisité: variables partagées

- **On définit $(S,i,E,F,Tran)=[\,[\,P_1 \,||\, \ldots \,||\, P_n\,]\,]$ par:**
  - $S=S_1 \times \ldots \times S_n \times S_g$,
  - $s_0=(s_{01},\ldots,s_{0n})$,
  - $F=F_1 \times \ldots \times F_n$
  - $E=E_1 \cup \ldots \cup E_n$,
  - $Tran=$
    - $\{ (s_1,\ldots,s_j,\ldots,s_n,s_g) \xrightarrow{tj} (s_1,\ldots,s'_j,\ldots,s_n,s_g) \mid s_j \xrightarrow{tj} s_j' \in Tran_j$ et t_j affectation de variable Locale $\}$
    - $\cup$
    - $\{ (s_1,\ldots,s_j,\ldots,s_n,s_g) \xrightarrow{tj} (s_1,\ldots,s_j,\ldots,s_n,s'_g) \mid s_j \xrightarrow{tj} s_j' \in Tran_j$ et t_j affectation sur une variable Globale $\}$
- **L'historique d'exécution d'un programme $P=P_1 \,||\, \ldots \,||\, P_n$**
  - L'ensemble fini de traces : $s0 \xrightarrow{t1} s1 \ldots \xrightarrow{tj} s_j \xrightarrow{tm} s_m$ où
  - $\forall\, s_{i-1} \xrightarrow{ti} s_i \in tran$
  - Et
  - $s_m \in F$

Revenons à Mrs and Mr Smith, mais avant

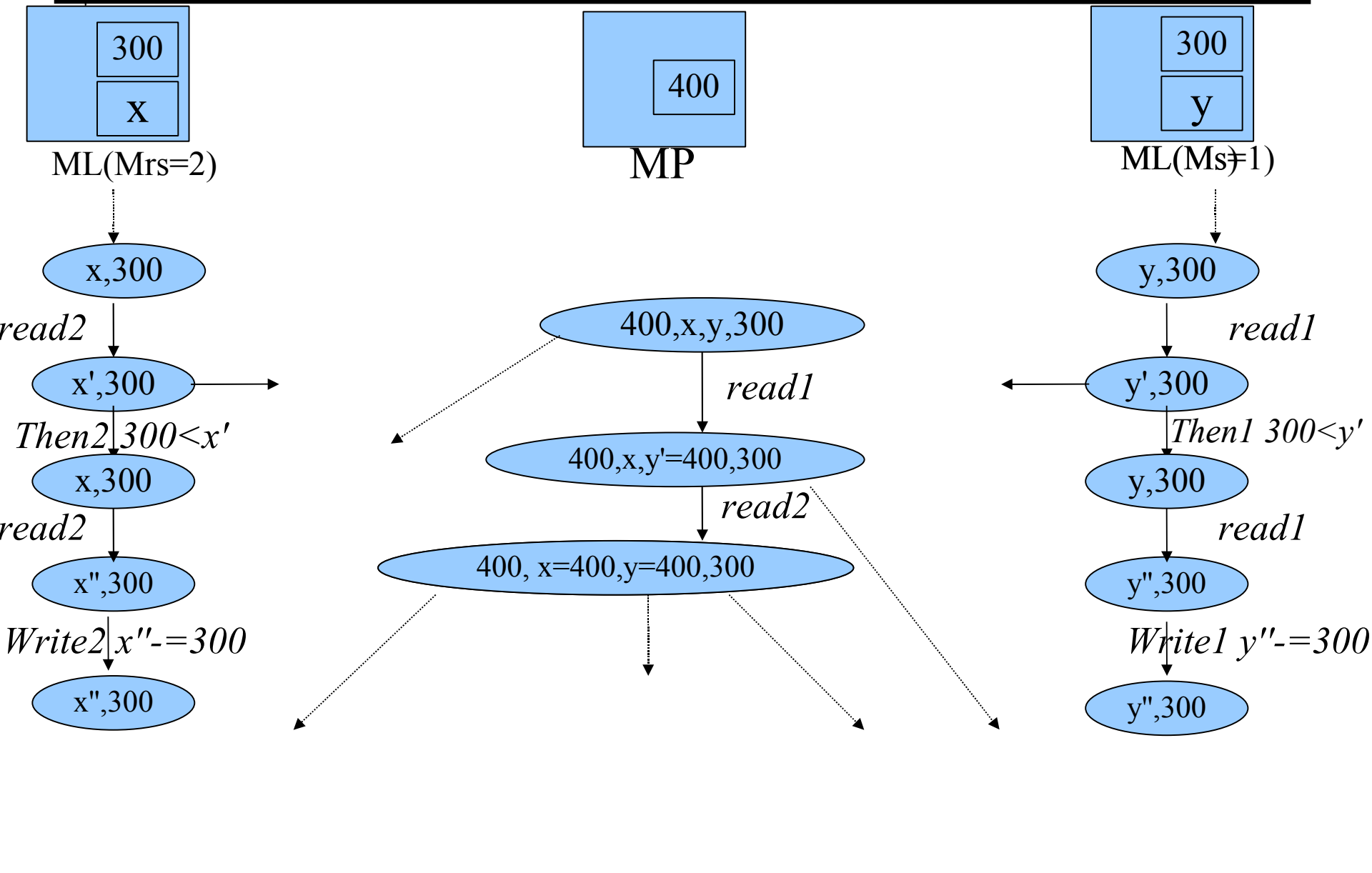
- **Pour comprendre ce qui se passe et s'assurer que notre langage et sémantique réprésente réellement le code de l'exemple il faut avant répondre à la question:**
  - Quel sont les actions atomiques (l'ensemble *E*)?
  - Comment la JVM traite les Processus? On a répondu a cette question c'est la sémantique par entrelacement (à quelque détail prés eg notion de priorité par exemple et technique d'ordonnancement mais ça sort du cadre de ce cours.)
- **La JVM : quelques généralités**
  - Chaque processus (threads) est chargé en mémoire avec une memoir local
  - Les variables partagés ici ***Compte*** sont logé dans la mémoir principal du processus maître, ***main()***.
  - Chaque processus possède une copie local de chaque variable partagée la cohérence de la mise à jours niveau bas est gérée par la JVM.

Actions atomiques de la JVM

- **Les actions atomique d'un thread**
  - *(U)se : transfère le contenu de la copie locale d'une variable à la machine exécutant le thread.*
  - *(A)ssign: transfère une valeur de la machine exécutant le thread à la copie locale d'une variable de ce même thread*
  - *(L)oad: affecte une valeur venant de la mémoire principale à la copie locale à un thread d'une variable*
  - *(S)tor : transfère le contenu de la copie locale à un thread d'une variable à la mémoire principale (qui va ainsi pouvoir faire un write)*
- **Les action de la mémoir principal : quelque généralités**
  - *(R)ead : transfère le contenu d'une variable de la mémoire principale vers la mémoire locale d'un thread (qui pourra ensuite faire un load)*
  - *(W)rite: affecte une valeur transmise par la mémoire locale d'un thread (par store) d'une variable dans la mémoire principale.*
- **NB: pour les types *double* et les *entier long* ces actions ne sont pas atomiques (deux *read* sont necessaires pour lire un double)**

```
public boolean retirer(int somme) {
    if (somme > 0)
        if (somme <= valeur) {
            valeur -= somme;
            return true;
        }
    return false;
}
```

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



la solution : synchronized

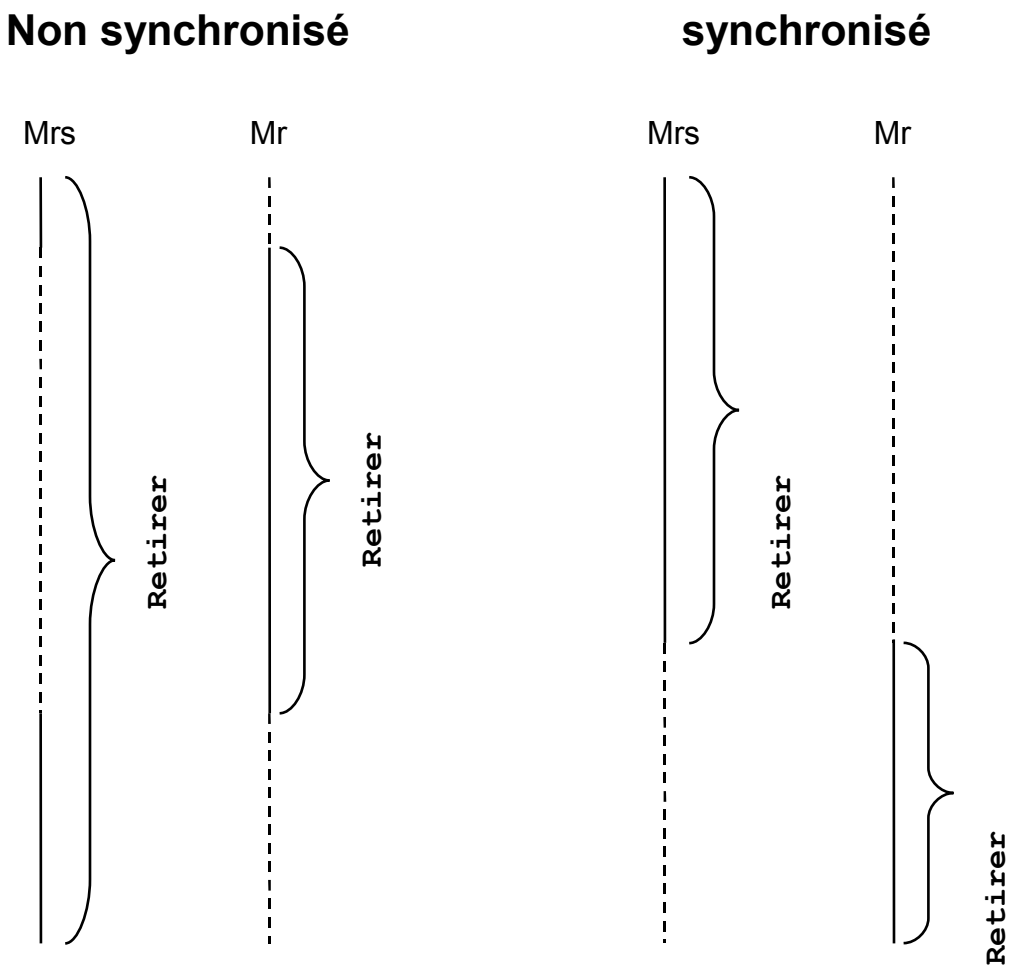
- **Problème**
  - nécessité d'accéder à des ressources partagées
    - ex. : accès à un objet et modification de son état
  - les threads s'exécutent de manière concurrente
  - selon l'ordre dans laquelle la donnée a été accédée, possibilité de corruption d'objets
  - nécessité de synchroniser l'accès
  - Dans notre cas garantir un accées à *retirer(...)* que ssi l'état de la valeur du compte est stable
  - Ceci par l'exclusion mutuel entre les exécutions de Retier
  - Donc on utilise une primitive Java qui rend l'exécution de Retirer atomique ce qui garantie son exlusion mutuel
- **Solution :**
  - => utilisation de *synchronized* signale que le code de cette methode pour chaque objet à un moment donée n'est éxécuté que par un seul thread.

Application sur notre exemple

```
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 Compte {
   private int valeur;

   Compte(int val) {
      valeur = val;
   }

   public int solde() {
      return valeur;
   }
   public synchronized  void depot(int somme) {
      if (somme > 0)
         valeur+=somme;
   }

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

Synchronisation des méthodes



Point de vue de la JVM

- **Rajouter deux actions atomiques aux threads.**
  - *(L)ock : réclame un verrou associé à une variable à la mémoire principale.*
  - *(U) nlock : libère un verrou associé à une variable*
- **Quand une méthode déclarer *synchronized*.**
  - A l'appel de cette méthode l'objet propriétaire est bloqué
  - Mais pas dans sa totalité : seule les variable global et local utilisé par l'ensemble des méthodes ***synchronized*** de l'objet sont verrouillées les autres reste utilisable.
  - Un appel d'une méthode *synchronized* d'un objet déjà verrouillé bloque le processus appelant.
  - Le processus est débloquées lorsque la méthode *synchronized* est terminer (correctement ou incorrectement)
- **Comment ça marche**
  - Chaque objet fourni un verrou
  - Mais aussi un mécanisme de mise en attente et notification.
  - *void wait()* : dit être appelé donc d'un bloque *synchronized.*
  - *Void notify() pemet de notifier un thred en attente d'une condition de l'arrivée de celle-ci sur l'objet :* dit être appelé donc d'un bloque *synchronized.*
  - *Void notifyAll().*

Intégration de mécanisme de verrou dans notre sémantique

- **On rajoute dans *L* deux actions**
  - *P(.)* : vérrouiller une variable.
  - *V(.)*: déverrouiller une variable.
- **Rajoutant également à chaque variable un verrou sous forme d'une fonction *k:N ---->{0,1}***
- **On rajoute à notre état *(l,H,k)* où *k* est état de vérouillage des variables (un vecteur binaire). K_0=<*1*,....,*1*>.**
- **La sématique opérationnelle P(.)**
  - $(P(x);l,H,K) \xrightarrow{p(x)} (l,H,k'=[k(x)=0](K))$ *si* $k(x)=1$
  - *P(x)* n'est possible que ssi la variable n'est pas verrouillé
  - **Par omission sur la condition ¬*(k(x)=1)* le processus reste bloqué.**
- **La sématique opérationnelle *V(.)***
  - $(V(x);l,H,K) \xrightarrow{V(x)} (l,H,k'=[k(x)=1](K))$
  - Remarquez aucune condition n'est associer à la libération d'une variable
- **=>Noté bien que c'est le bloquage des transitions de *P(.)* font que certain entrelacement deviennent impossible et donne l'impression de l'atomique.**

Retour sur les propritées de *synchronized*

- **A chaque appel d'une *synchronized***
- **La JVM appelle des *lock()* sur l'ensemble de variables utilisées, à la sortie elle appelle des *unlock()*.**
- **Quelques remarques**
  - Utilisation des *synchronized* provoque une serialisation des éxécutions
  - Conséquence : Une mauvaise utilisation provoque la baisse de performance.
  - Mécanisme de *synchronized* est un mécanisme décalaratif
  - => forcément statique
- **Besoin d'un mécanisme plus souple pour le verrouillage**
- **=>Impératif et non déclaratif**
- **Idée: pour les objets sujette à la consurrence utiliser un moniteur qui règlementera l'accées**
  - Utiliser le mécanisme de *synchronized* pour mettre en place des mécanisme dynamique de synchronisation par exclusion mutuel.
  - Quelqu'un a deviner de quoi je parle....???

Sémaphore : première forme binaire (1)

- **Rappel du principe**
  - Il y a Un jeton
  - Il y a des sections critiques qui peuevent être éxecuter de manière concurente par un ensemble de processus => besoin d'exclusion mutuel.
  - =>Seul celui qui possède le jeton peut exécuter les sections critiques
  - => permet une façon dynamique du passage de contrôle sur les sections critiques
  - C'est une implémentation de haut niveau de P(.) et V(.) sur une variable partagé ici le jeton (ou encore la sémaphore).

```
public class Semaphore {
   int n;

   public Semaphore() { n=1;  }

   public  void P( ) {
      if (n == 0) {
         try {
            wait(); // attendre que n des vient 1
         } catch(InterruptedException ex) {};
      }
      n--;
      System.out.println("P(jeton)");
   }

   public  void V() {
      n=1;
      System.out.println("V(jeton)");
      notify();
    }
   }
}
```

Rien ne vous choque???

```
public class Semaphore {
   int n;

   public Semaphore() { n=1;  }

   public  synchronized void P( ) {
      if (n == 0) {
         try {
            wait(); // attendre que n des vient 1
         } catch(InterruptedException ex) {};
      }
      n--;
      System.out.println("P(jeton)");
   }

   public  synchronized void V() {
      n=1;
      System.out.println("V(jeton)");
      notify();
    }
   }
}
```

Sémaphore : première forme binaire (2)

```
public class essaiPV extends Thread {
   static int x = 3;
   Semaphore u;

   public essaiPV(Semaphore s) {
      u = s;
   }
   public void run() {
      int y;

      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);

   }
   public static void main(String[] args) {
      Semaphore X = new Semaphore();
      new essaiPV(X).start();
      new essaiPV(X).start();
   }
}
```

Sans

Sans :

```
Thread-2: x=4
Thread-3: x=4
```

Avec :

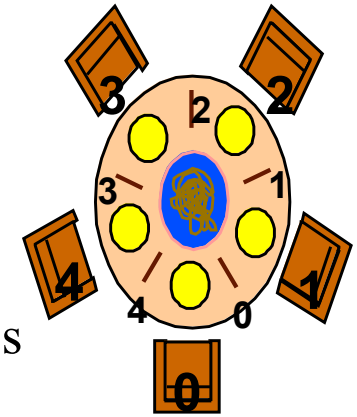
```
P(jeton)
Thread-2: x=4
V(jeton)
P(jeton)
Thread-3: x=5
V(jeton)
```

```
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();
      new essaiPV(X).start();
      new essaiPV(X).start();
   }
}
```

avec

Un classique : le dîner des Philosophes

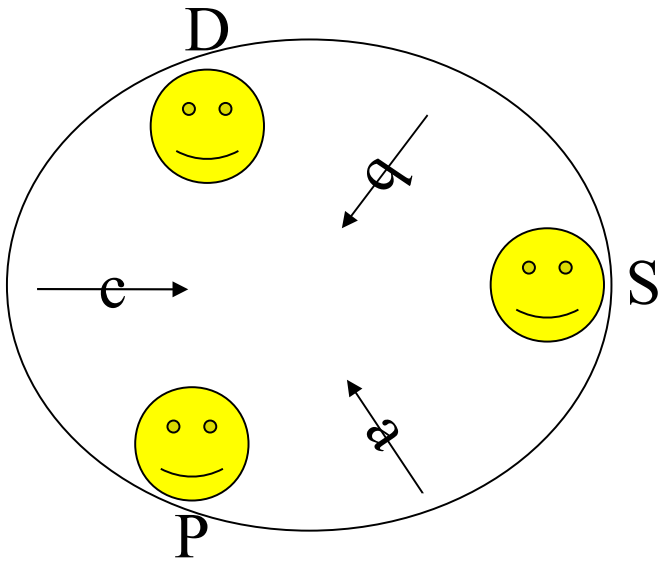
- **Le dîner des Philosophes :**
  - n Philosophes et n fourchettes
  - Assis sur une table ronde
  - Une fourchette entre deux philosophes.
  - Pour se servir des pâtes
  - un philosophe a besoin de deux fourchettes en même temps
  - Il est claire que on est dans un cas de ressource partagée



- **Problème d'algorthmique répartie.**
  - Règle d'or : il faut résoudre le problème par un algorithme unique pour chaque philosophe (processus)
  - => la symétrie est une propriété recherché car elle facilite l'analyse.
- **Un algorithme naïve serai serai que chaque philosophe :**
  1) Attendre que la fourchette gauche soit libre il la prend
  2) Attendre que la fourchette droite soit libre il la prend
  3) Mange
  4) Dépose la fourchette de gauche puis celle de droite.

Exercice

- **Utilisez les sémaphores pour programmer le dîner des philosophe.**
- **Un philosophe possède un nom ainsi que les deux fourchettes qui l'entourent**
- **Chaque fourchette est un jeton (sémaphore)**
- **Ecrivez également un main( ) pour un dîner entre :**
  - Platon
  - Descart
  - Spinoza



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(); // attendre que n des vient 1
            } catch(InterruptedException ex) {};
        }
        n--;
        System.out.println("P("+name+")");
    }
    public  synchronize void V() {
        n=1;
        System.out.println("V("+name+")");
        notify();
     }
    }
}
```

```
public class Phil extends Thread {
    Fork LeftFork;
    Fork RightFork;
    String names
    public Phil(Fork l, Fork 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 static void main(String[] args) {
        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("Descart");  Phil3.setName("Platon");
        Phil1.start();  Phil2.start(); Phil3.start();
    } }
```

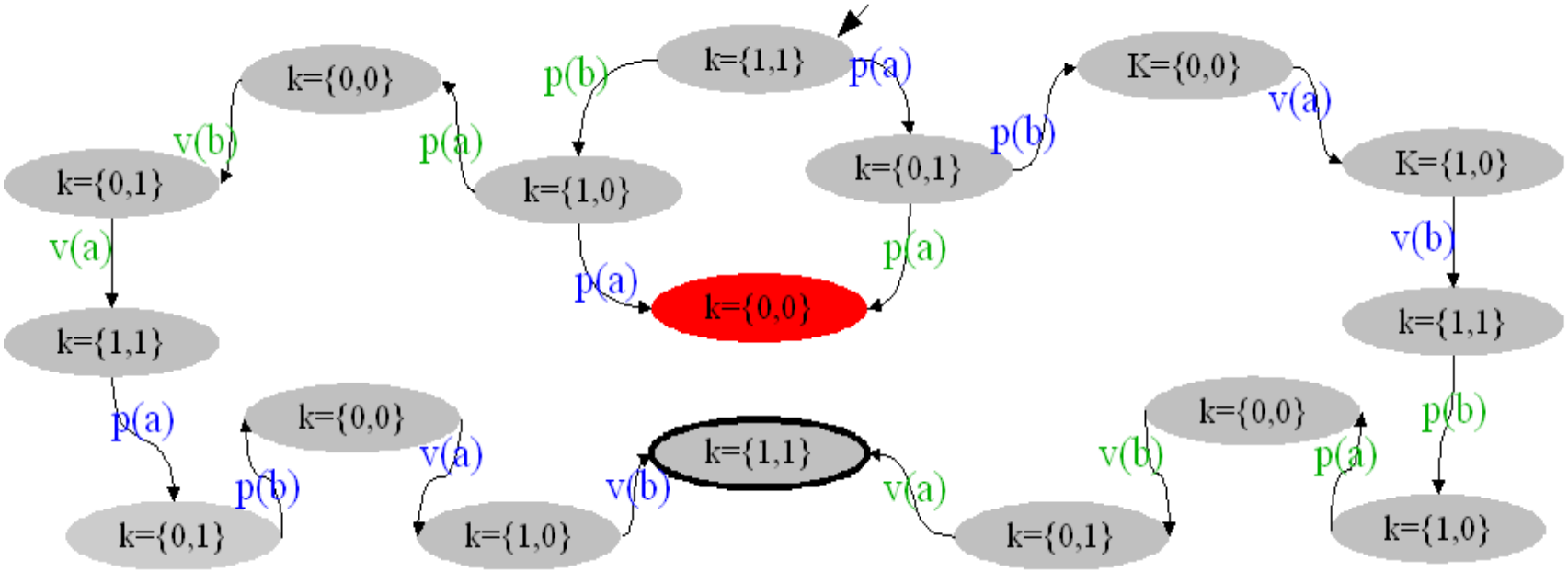
Y a un problème: le système bloc

- **Intuitivement :**
  - Dans une disposition circulaire la gauche de quelqu'un est la droite de son voisin de gauche.
  - Une exécution synchrone de la prise de la fourchette gauche=> aucune fourchette n'est libre.
  - Chaque philosophe possède une fourchette et il attend une deuxième
  - Aucun ne peux libérer la fourchette qu'il a, tantqu'il n' a pas trouver une deuxième => le système n'avance pas.
  - Ce qu'on appel un *deadlock*. (on verra plus loin un exemple de *livelock*)
- **Notion de blocage:**
  - Quand on peut synchroniser un ensemble de processus deux danger d'ordre logique c a d vos solution algorithmique peuvent vos poser des problèmes
  - Notion de bloquage more ou *deadlock*
    - L'application n'avance pas vers la solution (terminaison) car aucune action n'est desormes possible.
    - Sur le graph d'exécution

  - Notion de bloquage vivant ou livelock
    - L'application n'avance pas vers la solution (terminaison) seul des actions qui ramène à un cycle de la même configuration sont possible.
    - Sur le graph d'exécution

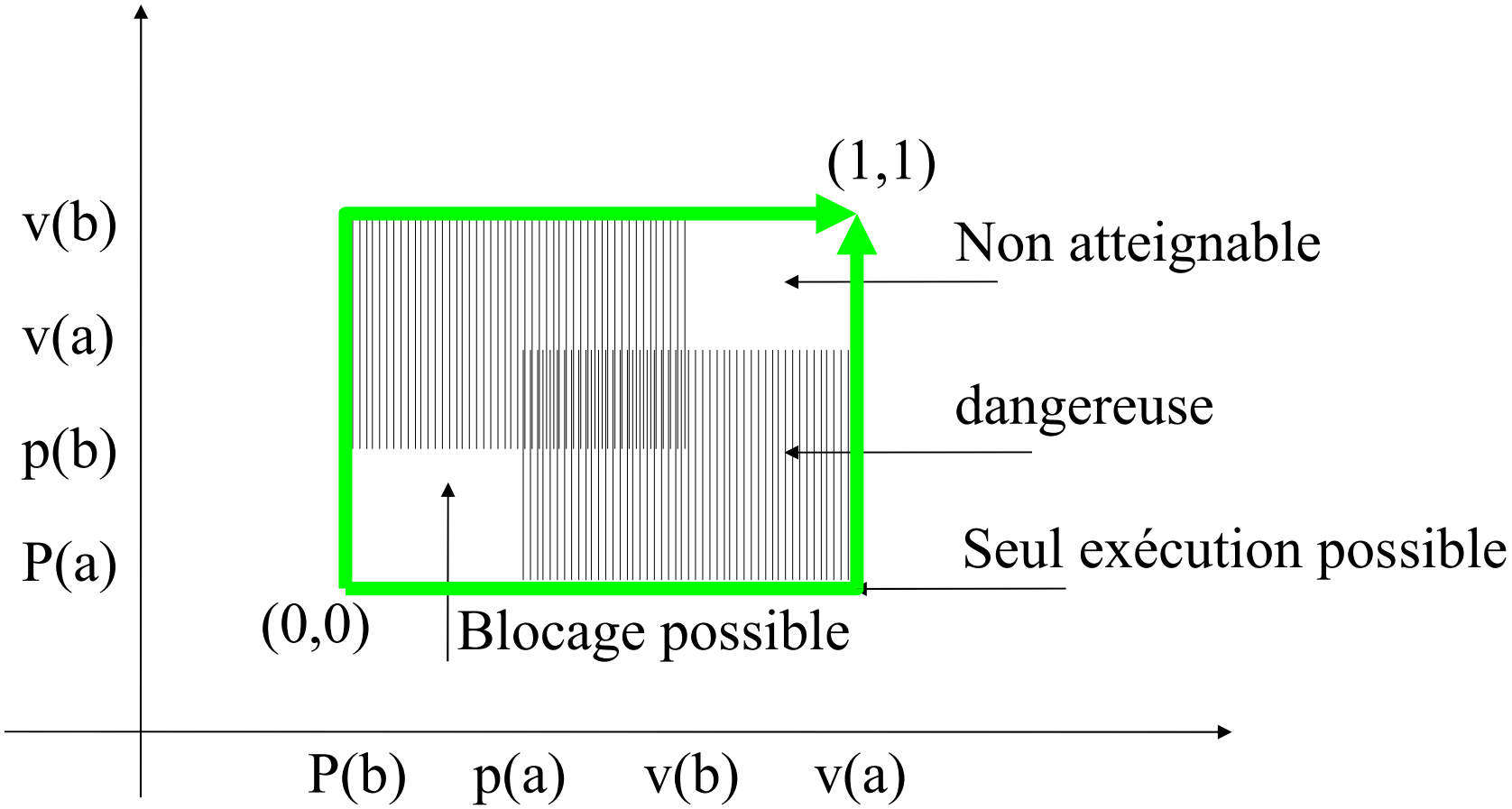
Caractérisation de la notion de blocage.

- **Revenons à notre sémantique.**
- **Pour comprendre ce qui s'est passé on a va prendre un cas simple.**
  - P1=p(a);p(b);v(a);v(b)
  - P2=p(b);p(a);v(b);v(a)
- **Nous sommes dans un cas où le parallélisme à une sémantique par entrelacement :**
- **Voici l'historique de P1||P2** ***(ici on représente seulement k)***



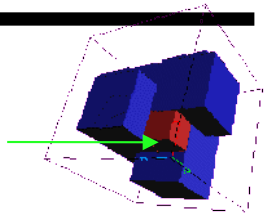
Graphiquement c'est encore mieux

- **Mettre les actions de chaque processus sur un axe**
- **Selon la sémantique par entrelacement:**
  - Les exécution possible sont représnetées par des courbes croissant allant de l'état initiale à l'état final (0,0) --->(1,1)

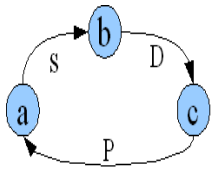


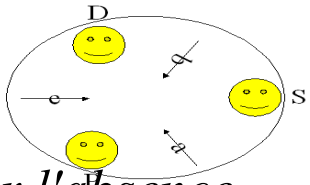
Complément sur le Bloquage

- **Graphe de notre solution pour le dîner des Philosophes :**

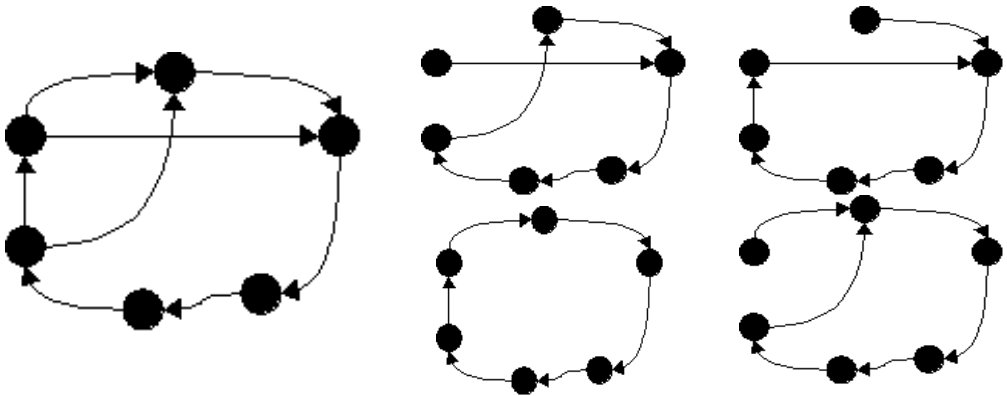


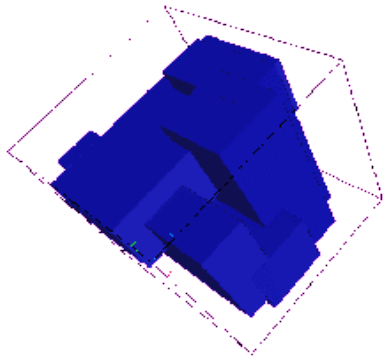
- **Le problème de blocage vient d'un ensemble de besoins ordonnés localement et circulaire globalement.**
- **On peut le voir si on dessine le graphe de requêtes sur les ressources.**
  - Un noeud par ressource
  - On a un arc d'un noeud *n* vers un noeud *m* si il existe un processus qui, ayant acquis un verrou sur *n* (sans l'avoir relaché), demande un verrou sur *m*





  - ***Attention!!***
    - *Son absence est est une condition suffisante pour prouver l'absence*
    - *Sa présence est non nécessaire pour prouver le blocage*





Verrous d'objets

- **Principe**
  - si un thread appelle une méthode synchronisée, l'objet devient verrouillé
  - analogie de la porte et de la clé
  - le gestionnaire de threads active périodiquement les threads qui attendent une clé
- **Remarques**
  - un thread qui quitte une méthode synchronisée sur une exception supprime le verrou
  - les autres threads sont toujours libres d'appeler les méthodes non synchronisées d'un objet verrouillé
    - ex. : méthode **`size()`**
  - un thread qui possède le verrou d'un objet a automatiquement l'autorisation d'accéder aux autres méthodes synchronisées de l'objet
  - un thread peut avoir plusieurs verrous
  - un verrou n'est possédé que par un seul thread

Wait et notify – Principes (1)

- **`wait`**
  - le thread actuel est bloqué et rend le verrou
  - le thread est placé dans une liste d'attente
  - tant qu'il est dans la liste d'attente, il n'a aucune chance d'être exécuté
- **`notify`** / **`notifyAll`**
  - **`notify`** supprime un thread choisi au hasard dans la liste d'attente
  - **`notifyAll`** supprime tous les threads dans la liste d'attente
  - un thread supprimé de la liste d'attente redevient exécutable
  - lorsque le verrou est à nouveau disponible, l'un des threads le prend et continue son exécution

Wait et notify – Principes (2)

- **Précautions d'emploi**
  - un thread qui appelle **`wait`** n'a aucun moyen de se débloquer lui-même
  - si tous les threads appellent **`wait`** sans qu'aucun n'appelle **`notify`**, on aboutit à un ***deadlock*** (verrous morts)
    - tous les threads sont bloqués
    - programme arrêté
    - aucun mécanisme en Java pour casser les verrous morts
  - les threads en attente **ne** sont **pas** réactivés si aucun thread ne travaille sur l'objet
  - lors d'un appel à **`notify`**, il est impossible de savoir quel thread sera débloqué
    - préférable d'utiliser **`notifyAll`**
  - utiliser **`notifyAll`** chaque fois que l'état d'un objet change d'une manière qui pourrait être avantageuse pour les threads en attente

Java.lang.Object

- **`void notifyAll()`**
  - déverrouille les threads qui ont appelé **`wait`** sur cet objet
  - ne peut être appelé qu'à partir d'une méthode synchronisée
  - déclenche une **`IllegalMonitorStateException`** si le thread courant n'est pas le possesseur du verrou
- **`void notify()`**
  - déverrouille un thread sélectionné au hasard parmi les threads qui ont appelé **`wait`** sur cet objet
  - idem
- **`void wait()`**
  - oblige un thread à attendre jusqu'à ce qu'il soit averti
  - idem

On a pas résolue le problème de nos Philosophes

- **Une idée???**
- **Sachant que :**
  - On doit garder la symétrie de l'algorithme
- **Indication :**
  - Les philosophe se distingue par leurs positions

- **Solution:**
  - Garder le même algorithme
  - Jouer sur la position
  - Chaque philosophe selon sa position :
    - Paire => prendre gauche puis droite.
    - Impaire => prendre droite puis gauche.
- **Exo:**
  - Dessinez graphe de requête?
  - Faite la preuve que l'application ne se bloque pas c a d au moins un Philosophe qui mange et un avancement est possible.

Sémaphore à compteurs :

- **Illustration sur un cas classique : producteur consommateur**
  - un producteur génère des données et les met dans une zone tompon
  - un consommateur lit et traite ces données
  - La zone a une taille fini.
- **Plusieurs problème de synchronisation**
  - Il ne faut pas lir et écrire en même temps (possibilité de lire des des données incompmete)
  - Il ne faut pas lire si il y a aucun message.
  - Il ne faut ecrire si la zone tompon est pleine.
- **Idée utiliser une sémaphore à compteur :**
  - La sémaphore binaire est un cas particulier de sémaphore avec compteur (1)
  - Pour déposer un message le producteur doit verrouiller le tompon en decrimentant

Code général d'une 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+"iemme)");
        notify();
         }
    }
}
```

Exercice:

- **Utilisez les sémaphores à compteur pour implémenter une communication entre deux processus Producteur et consommateur.**
- **Le producteur produit 10 fois plus vite que le consomateurs consomme.**
- **Écrivez le *main()* pour un producteur et un consomateur.**

solution(1)

```
public class Tompon {
  int taille;
  int n;
  int free=0;
  int next=0;
  Object [] tomp ;

  public Tompon(int max) {
      n = max;
      taille=max;
      tomp=new Object[n];
  }

  public synchronized void Produire(Object b) {
      if (n == 0) {
          try {
              wait();
          } catch(InterruptedException ex) {};
      }
      tomp[free]=b;
      free= (free+1)%taille;
      n--;
      System.out.println("P("+b+")");
      notify();
  }
```

```
public synchronized void consome( ) {
      if(n==taille)
        try {
            wait();
        } catch(InterruptedException ex) {};
      Object o=tomp[next];
      next=(next+1)%taille;
      System.out.println("V("+o+")");
      n++;
       notify();
  }
}
```

```
public static void main(String[] args) {
  Tompon t=new Tompon(5);
  Producer p1=new Producer(t);
  Consumer c1=new Consumer(t);
  c1.start();
 p1.start();
 }
```

solution(2)

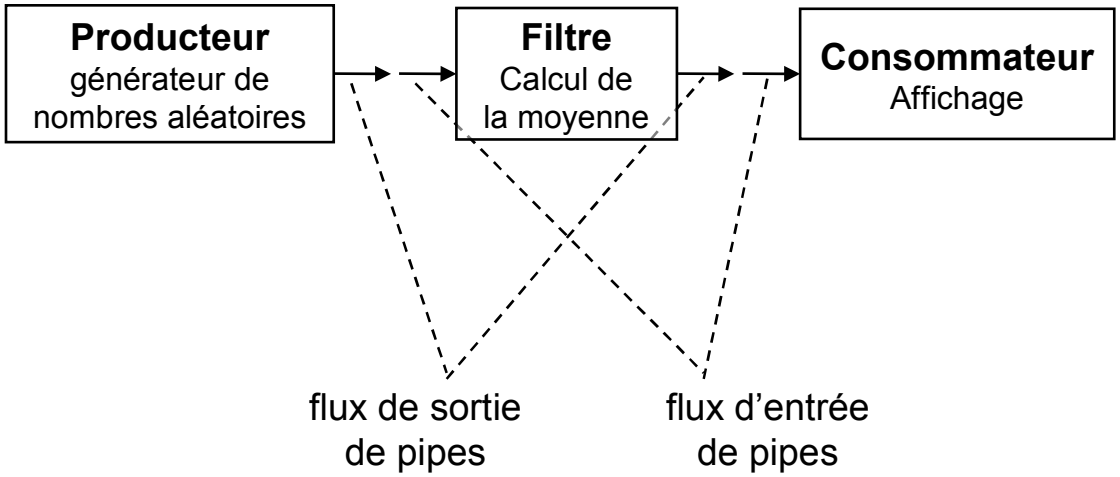
```
public class Consumer extends Thread {
  Tompon zone;
 public Consumer(Tompon t) {zone=t;}

 public void run()
 {int i=0;
  while(true)
 {zone.consome();
  try {
    this.sleep(5000);
   }
   catch (InterruptedException ex) {
   }
   i++;
}}}
```

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

 public void run()
 {int i=0;
  while(true)

 {zone.Produire("message"+i);
  try {
    this.sleep(50);
   }
   catch (InterruptedException ex) {
   }
   i++;
}}}
```

Communication entre threads

- **Exemple**
  - un producteur génère des données
  - un consommateur lit et traite ces données
- **Principe d'utilisation de pipes**
  - si aucune donnée disponible en lecture, le thread consommateur se bloque
  - si le producteur génère des données trop rapidement, l'opération d'écriture des données se bloque
- **En Java :**
  - **`PipedInputStream`**, **`PipedOutputStream`**
  - **`PipedReader`**, **`PipedWriter`**
- **Intérêt**
  - Conception simple et modulaire
  - connexion entre threads sans se préoccuper de leur synchronisation

Exemple


Producteur
générateur de
nombres aléatoires
Filtre
Calcul de
la moyenne
Consommateur
Affichage
flux de sortie
de pipes
flux d'entrée
de pipes

Exemple – code complet (1)

```
import java.util.*;
import java.io.*;

public class PipeTest {
  public static void main(String args[]) {
    try {
      // crée les pipes
      PipedOutputStream pout1 = new
                                PipedOutputStream();
      PipedInputStream pin1 = new
                              PipedInputStream(pout1);

      PipedOutputStream pout2 = new
                                PipedOutputStream();
      PipedInputStream pin2 = new
                              PipedInputStream(pout2);

      // crée les threads de traitement
      Producer prod = new Producer(pout1);
      Filter filt = new Filter(pin1, pout2);
      Consumer cons = new Consumer(pin2);

      // démarre les threads
      prod.start();
      filt.start();
      cons.start();
    }
    catch (IOException e){}
  }
}
```

```
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);
       }
     }
   }
}
```

Exemple – code complet (2)

```
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);
      }
    }
  }
}
```

```
class Consumer extends Thread {
  private double oldx = 0;
  private DataInputStream in;
  private static final double THRESHOLD = 0.01;

  public Consumer(InputStream is) {
    in = new DataInputStream(is);
  }

  public void run() {
    for(;;) {
      try {
        double x = in.readDouble();
        if (Math.abs(x - oldx) > THRESHOLD) {
          System.out.println(x);
          oldx = x;
        }
      }
      catch(IOException e) {
        System.out.println("Error: " + e);
      }
    }
  }
}
```

Java.lang.PipedInputStream

- **`PipedInputStream()`**
  - crée un nouveau flux d'entrée de pipe qui n'est pas encore connecté à un flux de sortie de pipe
- **`PipedInputStream(PipedOutputStream out)`**
  - crée un nouveau flux d'entrée de pipe qui lit ses données à partir d'un flux de sortie de pipe
- **`connect(PipedOutputStream out)`**
  - attache un flux de sortie de pipe pour lire des données

Java.lang.PipedOutputStream

- **`PipedOutputStream()`**
  - crée un nouveau flux de sortie de pipe qui n'est pas encore connecté à un flux d'entrée de pipe
- **`PipedOutputStream(PipedInputStream in)`**
  - crée un nouveau flux de sortie de pipe qui écrit ses données dans un flux d'entrée de pipe
- **`connect(PipedInputStream in)`**
  - attache un flux d'entrée de pipe pour écrire des données

Swing et les threads

- **Problème**
  - Swing n'est pas compatible avec les threads (en grande partie)
  - la plupart des méthodes ne sont pas synchronisées
    - pour des raisons d'efficacité
    - difficile à étendre
  - risque de corruption de l'interface utilisateur si des éléments de l'interface sont manipulés par plusieurs threads
- **Exemple**
  - un thread modifie une liste graphique
  - le thread de l'interface met à jour l'affichage au fur et à mesure des modifications dans la liste

Exemple

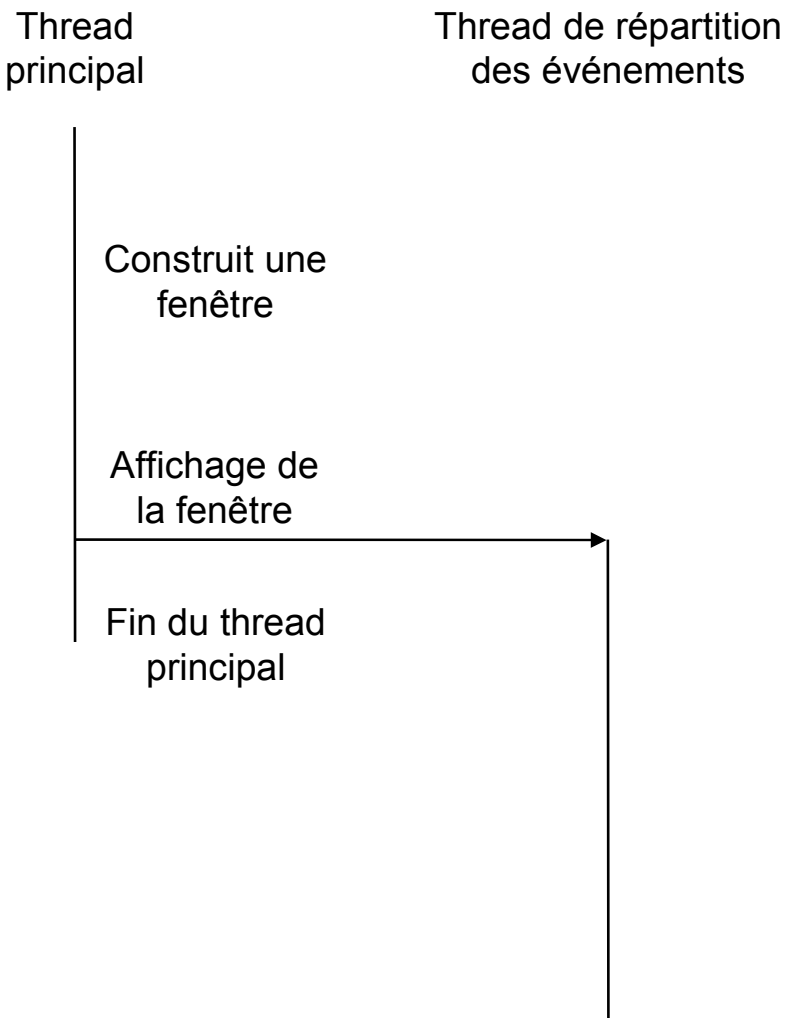
```
class BadWorkerThread extends Thread {
  private DefaultListModel model;
  private Random generator;

  public BadWorkerThread(DefaultListModel aModel) {
    model = aModel;
    generator = new Random();
  }

  public void run() {
    try {
      while (!interrupted()) {
        int i = Math.abs(generator.nextInt());
        if (model.contains(i))
          model.removeElement(i);
        else
          model.addElement(i);

        yield();
      }
    }
    catch (InterruptedException exception) {}
  }

}
```

Exécution de l'exemple

- quand une modification est faite dans la liste
  - un événement est généré pour mettre à jour l'affichage de la liste
- le thread d'affichage
  - lit la taille de la liste
  - commence à afficher les éléments de la liste
- si pendant ce temps, l'autre thread supprime des éléments de la liste
  - le thread d'affichage croit que la liste est plus grande que ce qu'elle n'est en réalité
  - le thread d'affichage va chercher à lire des valeurs en dehors des bornes de la liste et générer des exceptions **`ArrayIndexOutOfBounds`**

Les threads dans un programme Swing

- **Principe**
  - presque tout le code correspond à des gestionnaires d'événements
    - interaction avec l'interface
    - rafraîchissement de l'affichage
  - ce code est dans le thread de répartition des événements



Recommandations

- si une tâche nécessite beaucoup de temps, lancer le travail dans un nouveau thread
  - évite de monopoliser le thread de répartition des événements
  - l'application semblerait morte
- si une tâche peut être bloquée sur une entrée ou une sortie, lancer le travail dans un nouveau thread
  - l'interface utilisateur serait bloquée pendant la durée d'attente
- si vous devez attendre pendant une durée spécifiée, utiliser un chronomètre plutôt que de mettre le thread de répartition en sommeil
- le travail effectué dans vos threads ne doit pas interférer avec l'interface utilisateur (*règle du thread unique*) :
  - lire les infos dans l'interface avant de lancer les threads
  - mettre à jour l'interface à partir de l'interface de répartition une fois que vos threads sont terminés

Exceptions à la règle

- **Quelques méthodes sont compatibles avec les threads**
  - repérées dans l'API par le texte "*This method is thread safe, although most Swing methods are not*"
  - exemples
    - `JtextComponent.setText`
    - `JTextArea.insert`
    - `JTextArea.append`
    - `JTextArea.replace`
  - la méthode suivante peut être appelée par n'importe quel thread
    - `JComponent.repaint`
  - Ajouter et supprimer des écouteurs d'événements
  - Construire des composants, définir leurs propriétés, les ajouter dans des conteneurs
    - tant qu'aucun composant n'a été réalisé

Interaction entre threads et IHM

- **`invokeLater` et `invokeAndWait`**
  - fournies par la classe **`EventQueue`**
  - demandent à ce qu'un traitement soit effectué dans le thread de répartition
    - l'événement correspondant au traitement est envoyé dans la queue des événements
    - **`invokeLater`** se termine aussitôt traitement effectué de manière asynchrone
    - **`invokeAndWait`** attend jusqu'à ce que le traitement ait été effectué traitement effectué de manière synchrone
- **Procédure**
  - placer le code Swing dans la méthode run d'une classe qui implémente Runnable
  - créer un objet de cette classe
  - transmettre l'objet à la méthode **`invokeLater`** ou **`invokeAndWait`**

Exemple corrigé

Solution 1

```
class ListUpdater implements Runnable {
  DefaultListModel model;

  public ListUpdater(DefaultListModel aModel) {
    model = aModel;
  }

  public void run() {
    if (model.contains(i))
      model.remove(i);
    else
      model.addElement(i);
  }
}

class GoodWorkerThread extends Thread {
  DefaultListModel model;

  ...

  public void run() {
    while (!interrupted()) {
      int i = Math.abs(generator.nextInt());
      Runnable updater = new ListUpdater(model);
      EventQueue.invokeLater(updater);
    }
    yield();
  }
}
```

Solution 2 : classe interne anonyme

```
class GoodWorkerThread extends Thread {
  DefaultListModel model;

  ...

  public void run() {
    while (!interrupted()) {
      int i = Math.abs(generator.nextInt());
      EventQueue.invokeLater(new Runnable() {
        public void run() {
          if (model.contains(i))
            model.remove(i);
          else
            model.addElement(i);
        }
      }
      yield();
    }
  }
}
```

Java.awt.EventQueue

- **`static void invokeLater(Runnable runnable)`**
  - oblige la méthode run de l'objet runnable à être exécutée dans le thread de répartition des événements, après que événements en attente ont été traités
  - l'appel revient aussitôt
- **`static void invokeAndWait(Runnable runnable)`**
  - oblige la méthode run de l'objet runnable à être exécutée dans le thread de répartition des événements, après que événements en attente ont été traités
  - l'appel se bloque tant que la méthode **`run`** n'est pas terminée

Les systèmes temporisés

Qu'est-ce qu'un système temporisé?

- **Les exemples développés jusqu'à présent ne sont pas concernés par le passage du temps.**
- **Tout ce qui nous intéressait était le fait que l'ordre des actions corresponde à la notion d'ordre correct ou encore voulu.**
- **Les systèmes temporisés sont des systèmes dont le comportement est sensible au passage du temps.**
- **=> conséquence : leur correction dépends de l'exécution de certaines actions à des dates ou des temps spécifiques.**
- **Exemple : nous voulons mettre en place une classe système qui reconnait la différence entre un clic et un double-clic**
  - Un clic est un clic
  - Un double-clic est un clic qui a été précédé par un clic dans un laps d'unité de temps défini.

Quel type de système temporisé ?

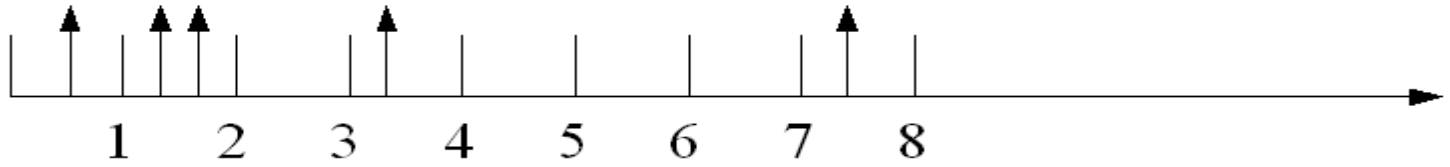
- **Le temps existe et passe aussi bien pour notre classe que pour l'utilisateur.**
- **Pour que la classe soit correcte il faut que le délai entre un clic et le second (pour faire un double clic) soit réalisable pour notre utilisateur.**
- **Le traitement du signal clic se fait sur le même système qui exécute notre classe**
  - => il ne faut donc pas que la durée de reconnaissance du signal soit plus grande que l'interval qui sépare les deux clics exigés par la classe.
- **On va donc faire les deux hypothèses suivantes :**
  - H1: une action a une durée nulle
  - H2: on n'assume pas qu'une infinité d'actions a une durée nulle
- **Une approche plus difficile et qui sort du cadre de ce cours est que :**
  - H: la durée d'une action n'excède pas une valeur de ce qu'on appelle dans les systèmes temps réel

Modelisation du temps : un temps discret

- **Dans le cours, on a dit que le modèle de concurrence est considré comme asynchrone.**
  - => les processus avancent à des vitesses arbitraires (aucune hypothèse sur la durée des actions).
- **Comment alors rendre un processus sensible au passage du temps et synchroniser son exécution avec le passage du temps?**
- **Réponse:**
  1) Avoir un modèle temps discret dans lequel le passage du temps est signalé par des tics succesifs d'une horloge globale.
  2) Les tics sont des événements traités par le programme comme ses actions.
  3) Utilisattion du système événement-écouteur

Quelques remarques sur le modèle temps discret (1)

- **Une trace d'exécution d'un programme temporisé est donc une suite d'action du système avec des actions speciales tic.**
  - a1,a2,tic,....aj...tic.....an
- **Est ce qu'une telle modélisation est correcte?**
- **Considérons ce modèle sur notre exemple (d=3)**



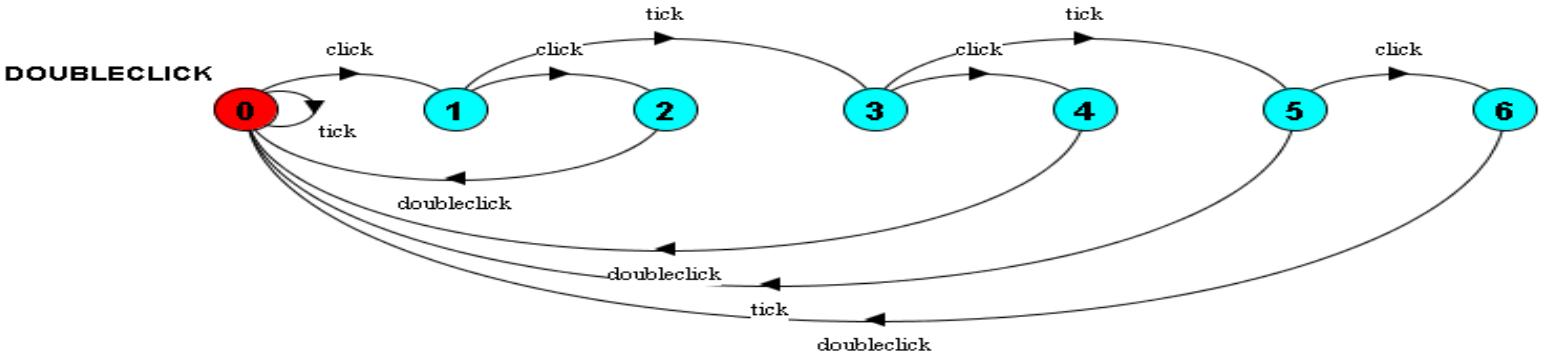
- **Quelle est la date d'un événement?**
  - Réponse: on ne sait pas avec certitude => quelque part dans [ ].
- **Est ce que le clic de période 7 est un double-clic par rapport à celui de la période 3?**
  - Réponse: au moins de 3 périodes => donc pas un double-clic
- **Dans un programme sequentiel temporisé la durée réelle qui sépare les evènements est (n±1)*T avec *n* le nombre de *tic* exécuté par le programme**

Quelques remarques sur le modèle temps discret (2)

- **Un modèle temps discret introduit une incertitude quant aux dates des événements**
- **Mais c'est une incertitude controlée dans un interval précis.**
- **Il faut noter également**
- **Que les machines fonctionnent sur ce modèle**
  - Top horloge qui règlemente l'exécution de nos programmes compatibles avec le modèle de l'environnment de notre programme.
- **Notez bien également que cette incertitude est inversement proportionnelle à T.**
- **On peut diminiuer selon notre besoin le temps qui sépare deux *tic***
- **Pour T=0.5 on aura des precisions du style : la durée qui sépare l'événement est de 2.5**

Retour à notre sémantique : impact de tic sur un programme séqunetiel

- **Rien ne change à part qu'un programe compte un nouveau élément parmi son alphabet.**
- **Sémantique opérationnelle pour un programme séquentiel**
  - (tic;P)---tic--->P
- **Exemple: Modélisation du programme double-clic pour D=3:**

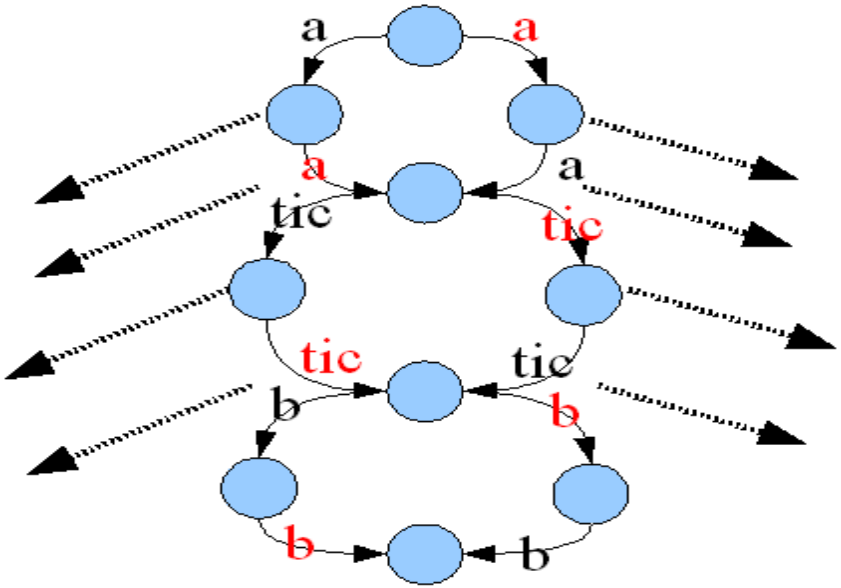


On appelle un programme temporisé, un programme qui contient dans l'une de ces traces possibles une ou plusieurs occurrences de l'action *tic*

Sémantique du tic dans un programme parallèle (1)

- **La question posée ici :**
  - *Est ce qu'on garde notre sémantique par entrelacement pour les actions tic?*
  - Ou, faut-il traiter de manière particulière l'arrivée d'un événement tic?
- **En premier lieu, on va supposer une sémantique par entrelacement et on va voir si ça garde la sémantique voulue**

a,tic,b
=>a[0,1]b

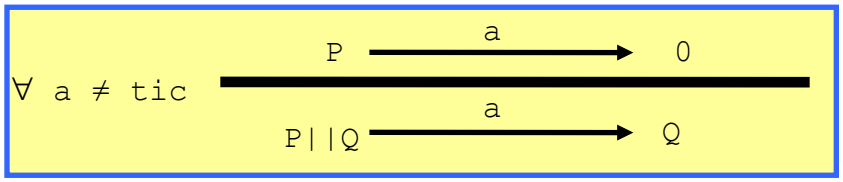


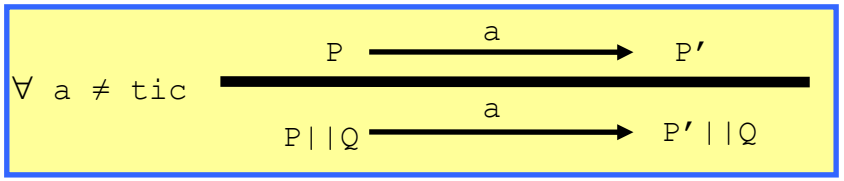
a,a,tic,tic,b,b
=>a[1,3]b
=>a[1,3]b

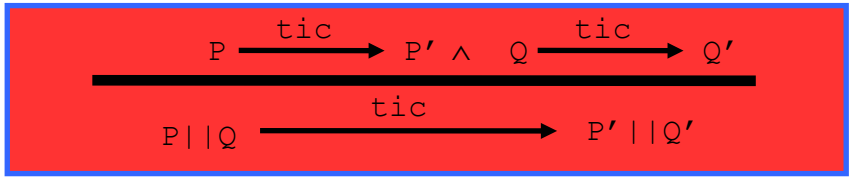
a,tic,b
=>a[0,1]b

Sémantique du tic dans un programme parallèle(2)

- **L'action *tic* n'est pas une action de chaque programme mais une réaction à une action de l'horloge càd le passage du temps.**
- **L'écoulement d'une période temps est le même pour tous les procesus**
- **Tous les tics (de chaque processus) doivent coincider dans le procesus global.**
- **Remodifions alors les règles de sémantique opérationelle du constructeur du parralélisme.**

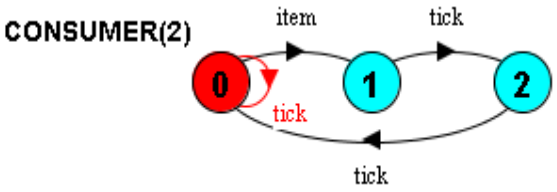


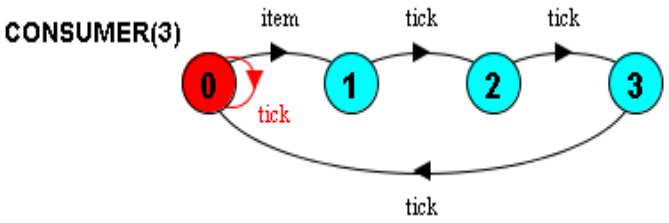


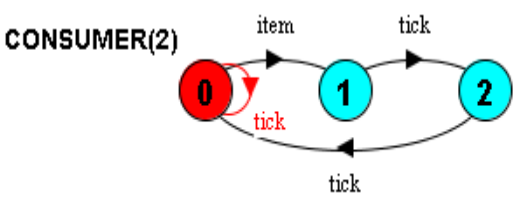


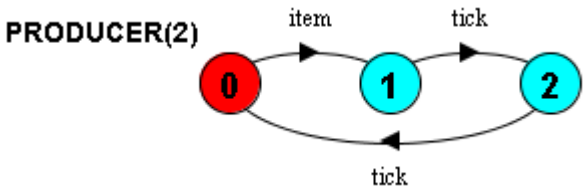
- **Un système peut évoluer de manière asynchrone => sémantique par entrelacement sauf**
- **Le temps est une action synchrone**
- **Donc ça sert à limiter les entrelacements donc synchroniser les processus**
- **Le temps ne passe que si l'ensemble des processus est dans un état de laisser passer le temps**
- **La raison pour que tout le monde comptabilise la même valeur même si ce n'est pas utile pour eux**
- **Par omission si un des processus ne peut plus laisser passer le temps alors l'ensemble des processus en état de le faire restent bloqués**
- **!!Faites attention à l'inconsistance temporelle de vos modèles exemple.**

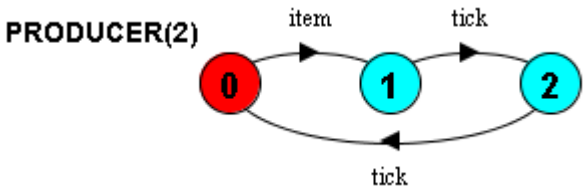
Illustration sur l'exemple de producteur consommateur

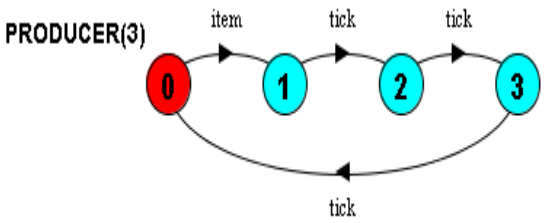








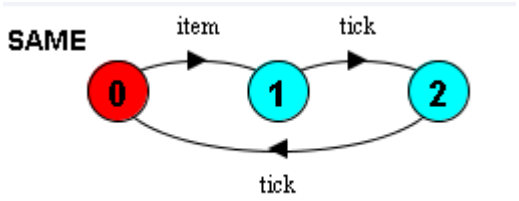


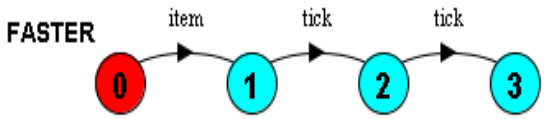


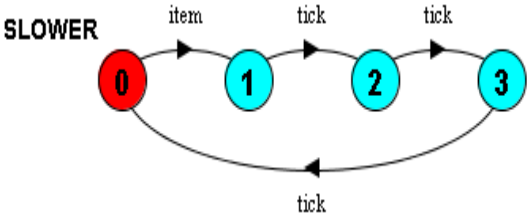
Producer(2) || consumer(2)

Producer(3) || consumer(2)

Producer(2) || consumer(3)





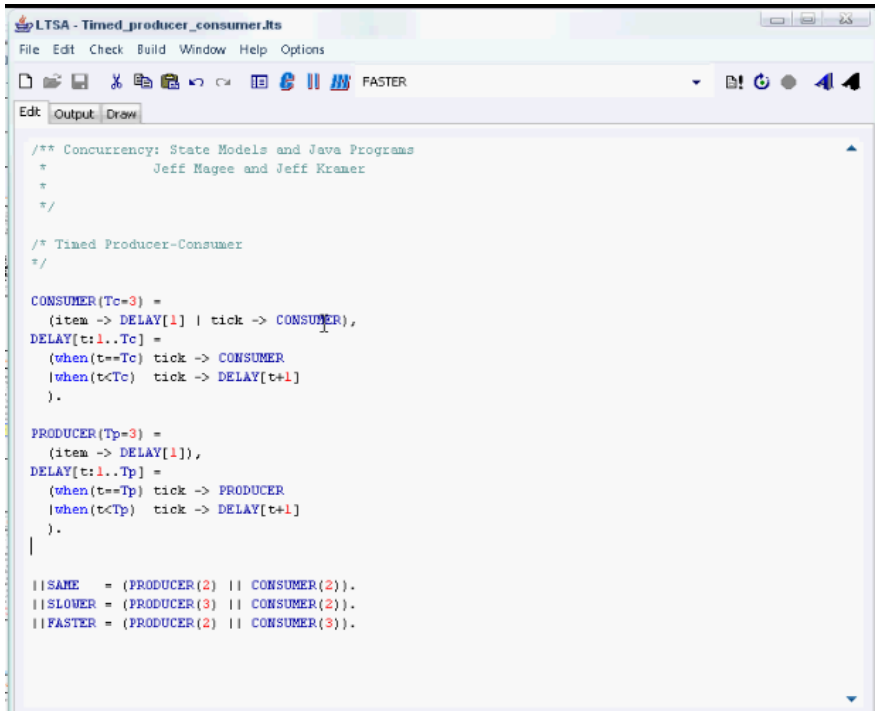


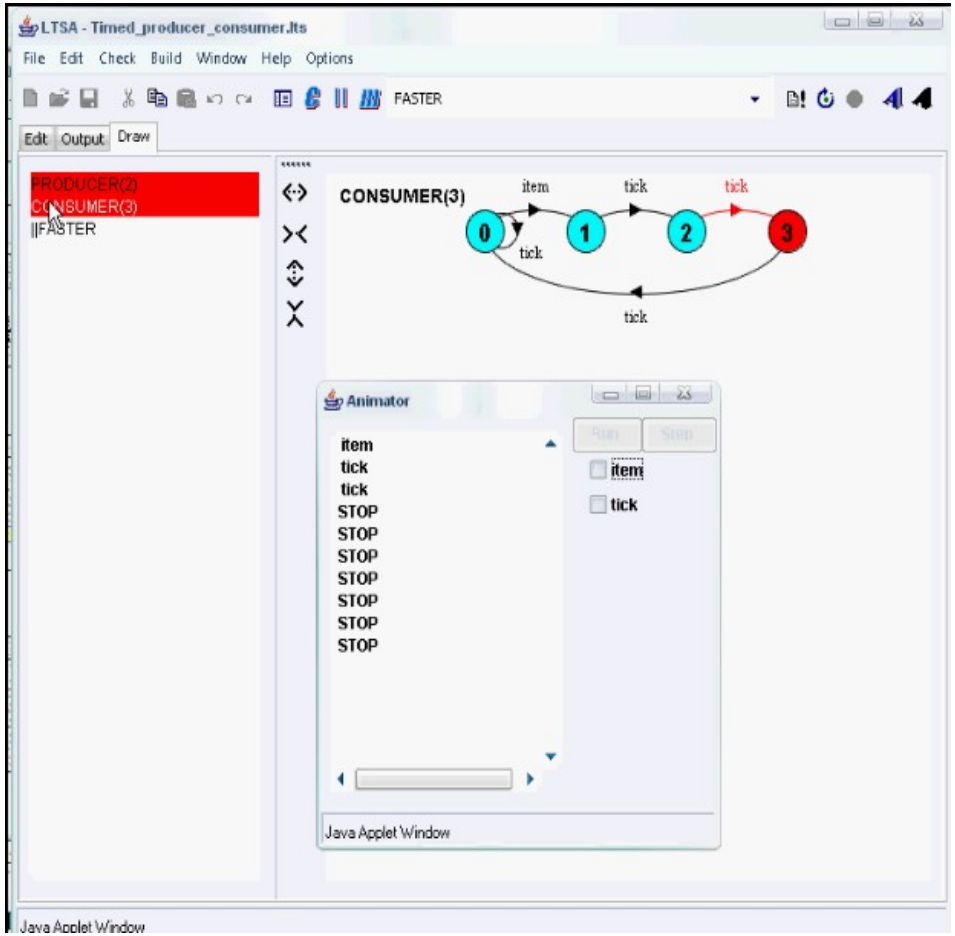
blocages

Attention ici on suppose que item n'est possible que si les deux peuvent la faire

L'outils LTSA modéliser et tester

- **On verra cet outil en TD**





class Timer et WorkTask

- **Java.util.Timer**
- **C'est une classe qui hérite de Thread qui permet de générer des événements à des intervalles de temps.**
- **A chaque interval de temps elle exécute un certain nombre d'instance WorkTask associé.**
- ***java.util.WorkTask***
- **Class abstraite qui implémente Runnable**

```
class MonAction extends TimerTask {
    int nbrRepetitions = 3;

    public void run() {
      if (nbrRepetitions > 0) {
        System.out.println("Ca bosse dur!");
        nbrRepetitions--;
      } else {
        System.out.println("Terminé!");
        t.cancel();
        }}}}
```

```
public class RepetAction {
  Timer t;
  public RepetAction() {
t = new Timer();
t.schedule(new MonAction(),
0,1*1000);
```

Une solution manuelle

- **Mettre en place une interface Timed pour désigner les processus temporisé.**
- **Deux méthode:**
- ***Public void pré-tic() throws unconsistancyTime***
  - Permet Finir les actions à exécuter avant le tic. Généralement utilisable pour definir le comportement entre les deux tic comme run de WorkTask
- ***Public void tic()***
  - definir le conportement une fois le tic et arrivé mettre un « timeout » a vrai ou initialiser un compteur...
- **Cérer une classe TimeManger comme timer qui prend une liste de timed et chaque intevall de temps**
  - Appel les pré-tic() des tous les objet Timed
  - Et ensuite les tic () de tous les Timed

Problème des solutions précédentes

- **Les actions d'entre les deux *tic* sont executés séquentiellement c a d pas d'entrelacement**
- **Pas de parallelism**
- **Solution :**
- **utiliser des processus pour implementer les *pre-tic()***
- **Mais attention !!!**
- **Il faut utliser *join()* dans la class *TimerManager* pour synchroniser la fin des pre-tic avnt de lancer les *tic()*.**

Quelques références

- **Ce cours a été réalisé en se basant sur:**
- **Cous en ligne de *Eric Goubault***

  http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/Cours05html/index.html
- **Un livre très intéressant :**

  Concurrency: State Models & Java Programs de *Jeff Magee & Jeff Kramer*
- ***les 10 premiers chapitres sont dispo en ligne ainsi que les slides qui vont avec (En Anglais).***

  http://www-dse.doc.ic.ac.uk/concurrency/