Systèmes Distribués (2)

Guillaume Hutzler
LaMI (Laboratoire de Méthodes Informatiques)
SyDRA (Systèmes Distribués, Réactifs et Adaptatifs)
hutzler@lami.univ-evry.fr

1

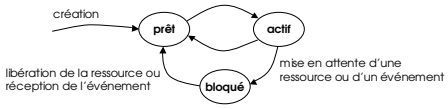
Plan du cours

2

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**



3

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

4

Politiques d'ordonnancement

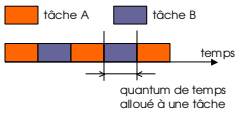
§ **Répartition manuelle**
- prévoir l'ensemble des contraintes temporelles
- définir un découpage des tâches
- définir une alternance des exécutions des phases qui respectent les contraintes du système de tâches

§ **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 »

5

Ordonnancement « interrupt-driven » (1)

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

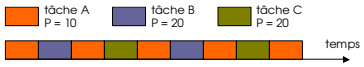


6

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



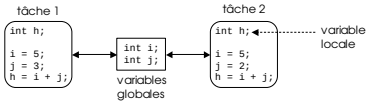
Les processus

§ **Un processus est défini par**
- son bloc de contrôle
  - priorité
  - point d'entrée
  - contexte (registres)
  - état
  - pile
- 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

Quelques problèmes liés à la concurrence

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



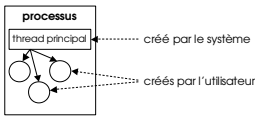
§ **Test & set**
- pb d'atomicité des opérations

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


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



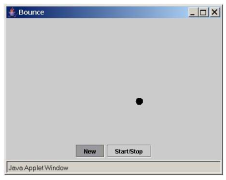
Threads vs. processus

§ **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 – code complet (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 – code complet (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) {}
  }
```

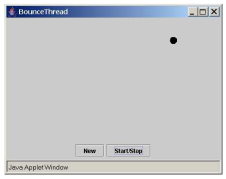
```
  public void move() {
    x += dx;
    y += dy;
    if (x < 0) {
      x = 0;
      dx = -dx;
    }
    if (x + XSIZE >= canvas.getWidth()) {
      x = canvas.getWidth() - XSIZE;
      dx = -dx;
    }
    if (y < 0) {
      y = 0;
      dy = -dy;
    }
    if (y + YSIZE >= canvas.getHeight()) {
      y = canvas.getHeight() - YSIZE;
      dy = -dy;
    }

    canvas.paint(canvas.getGraphics());
  }
}
```


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

- 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 – une balle qui rebondit version 2.0

- **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) {}
  }
  ...
}
```

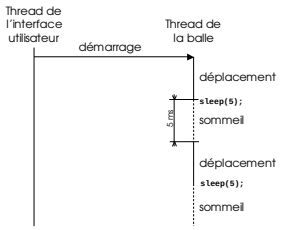
BounceThread – code modifié (2)

```
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);
    }
  }
}
```

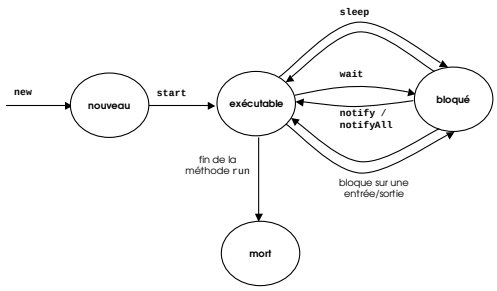
→

```
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);
    }
  }
}
```


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

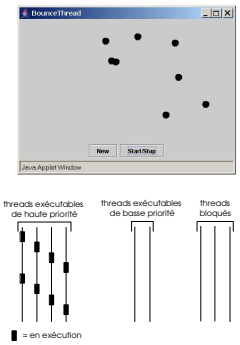
Etats d'un threads (3)

- **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

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 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) { canvas = c; }
  ...
}
```

```
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

Synchronisation

§ **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

§ **Exemple : gestion de comptes bancaire**
- Banque
  - 10 comptes
  - des transactions entre les comptes générées au hasard
    - une somme est prélevée sur un compte
    - elle est reversée sur un autre compte

Gestion de banque : code complet (1)

```
class Bank {
  public static final int NTEST = 10000;
  private int[] accounts;
  private long ntransacts = 0;

  public Bank(int n, double initialBalance) {
    accounts = new double[n];
    for (int i = 0; i < accounts.length; i++)
      accounts[i] = initialBalance;
  }

  public void transfer(int from, int to,
                       int amount) {
    if (accounts[from] < amount) return;
    accounts[from] -= amount;
    accounts[to] += amount;
    ntransacts++;
    if (ntransacts % NTEST == 0) test();
  }

  public void test() {
    double sum = 0;

    for (int i = 0; i < accounts.length; i++)
      sum += accounts[i];

    System.out.println("Transactions:" + ntransacts
           + " Sum: " + sum);
  }

  public int size() {return accounts.length;}
}
```

```
class TransferThread extends Thread {
  private Bank bank;
  private int fromAccount;
  private double maxAmount;
  private int REPS = 1000;

  public TransferThread(Bank b, int from,
                        double max)  {
      bank = b;
      fromAccount = from;
      maxAmount = max;
  }

  public void run() {
    try {
      while (!interrupted()) {
        for (int i = 1; i <= REPS; i++) {
          int toAccount = (int)(bank.size() *
                                Math.random());
          int amount = maxAmount * Math.random();
          bank.transfer(fromAccount, toAccount,
                        amount);

        }
        sleep(1);
      }
    }
    catch(InterruptedException e) {}
  }
}
```


Gestion de banque : code complet (2)

```
public class UnsynchBankTest {
  public static final int NACCOUNTS = 10;
  public static final double INITIAL_BALANCE =
    10000;

  public static void main(String[] args) {
    Bank b = new Bank(NACCOUNTS,
                      INITIAL_BALANCE);

      for (int i = 0; i < NACCOUNTS; i++) {
        TransferThread t = new TransferThread(b, i,
                                INITIAL_BALANCE);
        t.setPriority(Thread.NORM_PRIORITY + i%2);
        t.start();
    }
  }
}
```

§ **Trace d'exécution (Linux)**
- avec des transactions entières

```
Transactions:10005 Sum: 100000
Transactions:20002 Sum: 99991
Transactions:30000 Sum: 99986
Transactions:40003 Sum: 99986
Transactions:50003 Sum: 99986
Transactions:60001 Sum: 99986
Transactions:70003 Sum: 99986
Transactions:80003 Sum: 99986
Transactions:90003 Sum: 99986
Transactions:100003 Sum: 99986
```

- avec des transactions doubles

```
Transactions:10002 Sum: 100000.00
Transactions:20002 Sum: 100000.00
Transactions:30000 Sum: 100030.71
Transactions:40000 Sum: 100054.56
Transactions:50001 Sum: 100035.25
Transactions:60000 Sum: 100017.29
Transactions:70000 Sum: 94290.85
Transactions:80002 Sum: 95394.60
Transactions:90010 Sum: 95366.79
Transactions:100000 Sum: 95148.32
```


Problèmes de synchronisation

§ **Accès simultané à un compte**
`accounts[to] += amount`
- pb = instruction pas composée d'opérations atomiques
- décomposable en
  - charger **`accounts[to]`** dans un registre
  - ajouter **`amount`**
  - placer le résultat dans **`accounts[to]`**
- pb si un thread est interrompu entre la première et la troisième instruction
  - probabilité faible mais réelle

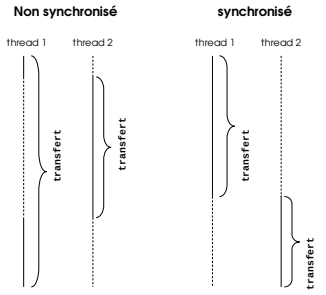
§ **Solution**
- empêcher l'interruption du thread pendant l'exécution de la méthode Bank.transfert
- déclarer la méthode comme étant **`synchronized`**
  - si un second thread appelle la méthode avant que le premier ait terminé son exécution, il est désactivé
  - il doit attendre la fin de l'exécution du premier thread

Gestion de banque : code modifié

```
class Bank {
  public static final int NTEST = 10000;
  private int[] accounts;
  private long ntransacts = 0;

  public Bank(int n, double initialBalance) {
    accounts = new double[n];
    for (int i = 0; i < accounts.length; i++)
      accounts[i] = initialBalance;
  }

  public void transfer(int from, int to,
                       int amount) {
    if (accounts[from] < amount) return;
    accounts[from] -= amount;
    accounts[to] += amount;
    ntransacts++;
    if (ntransacts % NTEST == 0) test();
  }

  public void test() {
    double sum = 0;

    for (int i = 0; i < accounts.length; i++)
      sum += accounts[i];

    System.out.println("Transactions:" + ntransacts
           + " Sum: " + sum);
  }

  public int size() {return accounts.length;}
}
```

```
class Bank {
  public static final int NTEST = 10000;
  private int[] accounts;
  private long ntransacts = 0;

  public Bank(int n, double initialBalance) {
    accounts = new double[n];
    for (int i = 0; i < accounts.length; i++)
      accounts[i] = initialBalance;
  }

  public synchronized void transfer(int from,
                       int to, int amount) {
    if (accounts[from] < amount) return;
    accounts[from] -= amount;
    accounts[to] += amount;
    ntransacts++;
    if (ntransacts % NTEST == 0) test();
  }

  public void test() {
    double sum = 0;

    for (int i = 0; i < accounts.length; i++)
      sum += accounts[i];

    System.out.println("Transactions:" + ntransacts
           + " Sum: " + sum);
  }

  public int size() {return accounts.length;}
}
```


Synchronisation de méthodes



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 à d'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 - Exemple (1)

- éviter de transférer de l'argent si solde du compte d'origine insuffisant

```
if (bank.getBalance(from) >= amount)
  bank.transfer(from, to, amount);
```

- pb : éviter d'interrompre le thread entre le test et le transfert

```
public synchronized void transfer(int from, int to, int amount) {
  while (bank.getBalance(from) < amount) {
    // attend
  }
  // transfère les fonds
}
```

- pb : le thread qui prend le verrou empêche tous les autres de déposer des fonds sur le compte

Wait et notify - Exemple (2)

§ **Solution**
- libérer le verrou et se placer en attente dans la méthode synchronisée

```
public synchronized void transfer(int from, int to, int amount)
      throws InterruptedException {
  while (accounts[from] < amount)
    wait();
  accounts[from] -= amount;
  accounts[to] += amount;
  notifyAll();
}
```


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

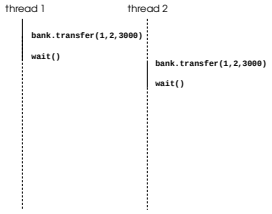
§ **`notify` / `notifyAll`**
- **`notify`** supprime un thread choisi au hasard dans la liste d'attente
- **`notifyAll`** supprime tous les threads dans la liste d'attente
- un thread supprimé de la liste d'attente redevient exécutable
- lorsque le verrou est à nouveau disponible, l'un des threads le 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***
  - 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

Exemple de deadlock (1)

- Compte 1 : 2000€
- Compte 2 : 3000€
- Thread 1 : transfert 3000€ du compte 1 vers le compte 2
- Thread 2 : transfert 4000€ du compte 2 vers le compte 1



Exemple de deadlock (2) - Utilisation de notify

- Situation initiale
  - Compte 1 : 19000€
  - Comptes 2 à 10 : 9000€
  - Thread 1 : transfert 9500€ du compte 1 vers le compte 2
  - Thread 2 à 10 : transfert 9100€ du compte i vers un autre compte
- le thread 1 exécute son transfert
- les autres appellent **`wait`**
- nouvelle situation
  - Compte 1 : 9500€
  - Compte 2 : 18500€
  - Comptes 3 à 10 : 9000€
- le thread 1 appelle **`notify`** et débloque le thread 3
- le thread 3 appelle à nouveau **`wait`**
- le thread 1 veut transférer 9600€ du compte 1 vers le 2
- il appelle **`wait`**
- **deadlock**

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
- déclence 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

L'interface `Runnable`

§ **Problème**
- si l'on veut transformer en thread une classe qui étend déjà une autre classe
  - ex. : ajouter un thread dans une applet (qui hérite déjà de **`Applet`**)
- héritage multiple impossible
- obligation de passer par une interface

§ **`Runnable`**
- créer une classe qui implémente l'interface **`Runnable`**
- créer un objet **`Thread`** pour exécuter le thread
  - fournir au thread une référence vers l'objet **`Runnable`** dans son constructeur
  - appeler la méthode **`start()`** de l'objet **`Thread`**
  - le thread appelle la méthode **`run()`** de l'objet **`Runnable`**

Exemple

```
public class Animation extends Applet implements Runnable {
  private Thread runner;
  ...

  public void start() {
    if (runner == null) {
      runner = new Thread(this);
      runner.start();
    }
  }

  public void run() {
    ...
  }
}
```


Et pourquoi pas...

```
class AnimationThread extends Thread {
  public void run() {
    ...
  }
}

public class Animation extends Applet {
  private Thread runner;
  ...

  public void start() {
    if (runner == null) {
      runner = new AnimationThread();
      runner.start();
    }
  }
}
```

- propre et simple...
- ... mais ne permet pas de conserver un accès aux données privées de l'applet

Java.lang.Thread

§ **`Thread(Runnable target)`**
- construit un nouveau thread qui appelle la méthode run() de la cible spécifiée

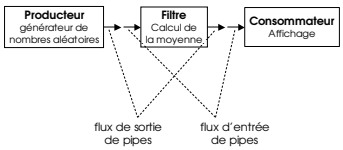
Java.lang.Runnable

§ **`void run()`**
- surcharger cette méthode pour y placer le code à exécuter par le thread

Communication entre threads

- **Exemple**
  - un producteur génère des données
  - un consommateur lit et traite ces données
- **Principe d'utilisation de pipes**
  - si aucune donnée 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



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

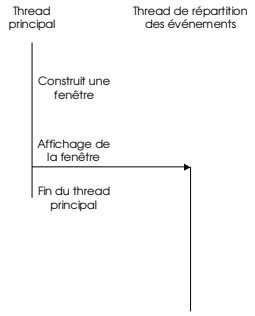
```
class BadWorkerThread extends Thread {
  private DefaultListModel model;
  private Random generator;

  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 peuvent être appelées 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 nonyme

```
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