ESSI 2
Programmation concurrente
TD2 : sémaphores et moniteurs en Java

JF Lalande, Michel Cosnard, Fabrice Peix

26 Avril 2006
inspiré des excellents TDs de Jean-Paul Rigault

1 Le coiffeur dormeur

Il s'agit encore d'un de ces problèmes de synchronisation mis sous une forme *"plaisante"*. Mais celui-ci est encore plus sérieux que le problème des philosophes, car on en trouve une application presque directe dans certains mécanismes des systèmes d'exploitation (comme l'ordonnancement des accès disque).

Un coiffeur possède un salon avec un siège de coiffeur et une salle d'attente comportant un nombre fixe F de fauteuils.

S'il n'y a pas de client, le coiffeur se repose sur son siège de coiffeur. Si un client arrive et trouve le coiffeur endormi, il le réveille, s'assoit sur le siège de coiffeur et attend la fin de sa coupe de cheveux. Si le coiffeur est occupé lorsqu'un client arrive, le client s'assoit et s'endort sur une des chaises de la salle d'attente ; si la salle d'attente est pleine, le client repasse plus tard. Lorsque le coiffeur a terminé une coupe de cheveux, il fait sortir son client courant et va réveiller un des clients de la salle d'attente. Si la salle d'attente est vide, il se rendort sur son siège jusqu'à ce qu'un nouveau client arrive.

Le but de de ce TD est d'associer une thread au coiffeur ainsi qu'à chaque client et de programmer une séance de coiffeur dormeur en Java... Nous allons essayer d'écrire deux versions de cet exercice, une avec les sémaphores, une avec les moniteurs.

1.1 Avec des sémaphores

Pour cet exercice et un peu en avance par rapport au cours, on utilisera les sémaphores de la JDK 1.5 disponibles dans */net/opt/sun-jdk-1.5.0/*. On écrira une classe pour le coiffeur (`HairDresser`) et une classe pour les clients (`Customer`). L'idée est que la communication se fasse au travers de sémaphores, bloquant l'exécution des threads quand cela est nécessaire.

Solution:
../java/BarberSemaphore.java

```
import java.util.concurrent.Semaphore;
```

```
// ============================================================================
// BarberSemaphore in Java
// Fabrice Peix --- ESSI --- 2006
// ----------------------------------------------------------------------------
// Usage:
// javac BarberSemaphore.java
// java BarberSemaphore nbChairs nbCustomers
// ============================================================================

/**
 * Simulate barber behaviors with Semaphore.
 *
 * @author Fabrice Peix
 */
public class BarberSemaphore extends Thread {

  /** Semaphores */
  private Semaphore busyChairsC, barberChairC;

  public BarberSemaphore(Semaphore busyChairs, Semaphore barberChair) {
    this.busyChairsC = busyChairs;
    this.barberChairC = barberChair;
  }

  /*
   * (non-Javadoc)
   *
   * @see java.lang.Runnable#run()
   */
  public void run() {
    while (true) {
      try {
        System.out.println("waiting for customer ...");
        busyChairsC.acquire();
        System.out.println("Customer arrive ! go to work...");
        barberChairC.release();
        Thread.sleep(1000);
      } catch (Exception e) {
        System.out.println("haircut interrupted");
      }
      System.out.println("Haircut is over");
    }
  }

  /**
   * Simulate custumer behaviors with Semaphore.
   *
   * @author Fabrice Peix
   */
  public class CustomerSemaphore extends Thread {

    private Semaphore freeChairs_, busyChairs_, barberChair_;
```

```
  public CustomerSemaphore(String name, Semaphore freeChairs,
      Semaphore busyChairs, Semaphore barberChair) {
    super(name);
    this.freeChairs_ = freeChairs;
    this.busyChairs_ = busyChairs;
    this.barberChair_ = barberChair;
  }

  /*
   * (non-Javadoc)
   *
   * @see java.lang.Runnable#run()
   */
  public void run() {
    try {
      while (!freeChairs_.tryAcquire())
        // Try to take a place in barbershop
        sleep(5000); // Go to pub drink a cup of coffee
      System.out.println(getName() + " Cool ! get a place");
      busyChairs_.release();
      System.out.println(getName() + "Waiting for barber...");
      barberChair_.acquire();
      freeChairs_.release();
    } catch (Exception e) {
      e.printStackTrace();
      System.out.println(getName() + "Interrupted during haircut");
    }
  }
}

public static void main(String[] args) {

  // read command-line arguments
  if (args.length != 2) {
    System.err.println("usage: BarberSemaphore nbChairs nbCustomers");
    System.exit(1);
  }
  int nbChairs = Integer.parseInt(args[0]);
  int nbCustomers = Integer.parseInt(args[1]);
  System.err.println("Starting BarberSemaphore with " + nbChairs
      + " chairs and " + nbCustomers + " customers");

  // Initialize Semaphore
  Semaphore freeChairs = new Semaphore(nbChairs);
  Semaphore busyChairs = new Semaphore(0);
  Semaphore barberChair = new Semaphore(0);

  // Create barber thread
  BarberSemaphore c = new BarberSemaphore(busyChairs, barberChair);
  c.start();

  // Create customers threads
  for (int i = 0; i < nbCustomers; i++) {
```

```
      CustomerSemaphore customer = c.new CustomerSemaphore("Client " + i + ":",
          freeChairs, busyChairs, barberChair);
      customer.start();
    }
  }
}
```

1.2 Avec des moniteurs

Pour utiliser les moniteurs, on créera une classe pour le salon de coiffeur. Cette classe Java `HairDresserShop` réalise sous forme d'un moniteur [1] la synchronisation de la boutique du coiffeur.

Votre classe devra contenir les méthodes nécessaires aux deux threads `HairDresser` et `Customer` :

- `Customer` utilise la méthode `goToHairDresser()` en lui passant son nom (ou son numéro) ; cette méthode retourne `true` si le client s'est effectivement fait coiffer ;
- `HairDresser` utilise la méthode `getCustomer()` pour traiter le client suivant (ou attendre s'il n'y a pas de client) et la méthode `showCustomerOut()` pour faire sortir le client lorsque la coupe est terminée.

Solution:

../java/BarberMonitor.java

```
import java.util.Vector;

//=========================================================================
//BarberMonitor in Java
//Fabrice Peix --- ESSI --- 2006
//-------------------------------------------------------------------------
//Usage:
//javac BarberMonitor.java
//java BarberMonitor nbChairs nbCustomers
//=========================================================================

/**
 * Simulate the behaviors of barber in Monitor case.
 *
 * @author Fabrice Peix
 */
class Barber extends Thread {
  /** the shop the barber works at */
  private BarberMonitor shop;

  /** Time of one cut */
  private final int cutTime = 2000;

  /**
   * Create a barber in a given shop.
```

[1] c'est-à-dire avec des méthodes synchronisées et l'utilisation des primitives `wait()` et `notify()/notifyAll()`

```
   *
   * @param shop
   *          The shop in which the barber is created
   */
  Barber(BarberMonitor shop) {
    super("barber");
    this.shop = shop;
  }

  /**
   * Method simulating hair cuting.
   */
  private void hairCut() {
    System.out.println("Barber cutting");
    try {
      sleep(cutTime);
    } catch (Exception e) {
    }
  }

  /*
   * (non-Javadoc)
   *
   * @see java.lang.Runnable#run()
   */
  public void run() {
    System.out.println("Barber at work ");
    while (true) {
      try {
        shop.getCustomer();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      hairCut();
      shop.showCustomerOut();
    }
  }
}

/**
 * Simmulate Customer behavior.
 *
 * @author Fabrice Peix
 */
class Customer extends Thread {
  /** Id of this customer */
  private int cust;

  /** The barber shop */
  private BarberMonitor shop;

  /**
   * Initialize a new customer.
```

```
   *
   * @param cust
   *          the id of this new customer.
   * @param shop
   *          the baber shop
   */
  Customer(int cust, BarberMonitor shop) {
    super("customer " + cust);
    this.cust = cust;
    this.shop = shop;
  }

  /*
   * (non-Javadoc)
   *
   * @see java.lang.Runnable#run()
   */
  public void run() {
    System.out.println("Customer " + cust + " starting");
    try {
      while (!shop.goToHairDresser()) {
        System.out.println("Customer " + cust + " still needs an hair cut");
        sleep(3000);
      }
    } catch (InterruptedException e1) {
      e1.printStackTrace();
    }
  }
}

/**
 * Simulate the barbershop behaviors. All concurrent access are managed in this
 * class.
 */
class BarberMonitor {

  /** Number of chairs in waiting room */
  int nbChairs;
  /** FIFO of customer present in the room */
  private Vector<Thread> customer = new Vector<Thread>();


  public BarberMonitor(int nbChairs) {
    this.nbChairs = nbChairs;
  }

  /**
   * Try to go to the barbershop.
   *
   * @return true if customer make an haircut and false otherwise.
   */
  public boolean goToHairDresser() throws InterruptedException {
    Customer current = (Customer) Thread.currentThread();
```

```
  synchronized (current) {
    synchronized (this) {
      if (customer.size() == nbChairs)
        return false;
      System.out.println(Thread.currentThread().getName() + " waiting");
      if (customer.size() == 0)
        notify();
      customer.add(current);
    }
    current.wait();
  }
  return true;
}

/** Save the last customer, used to print his name in ShowCustomerOut */
private Thread lastCustomer;

/**
 * Make an haircut or wait for customers.
 */
public void getCustomer() throws InterruptedException {
  // wait for customer
  synchronized (this) {
    System.out.println("Barber waiting for customer");
    while (customer.size() == 0)
      wait();
    lastCustomer = customer.remove(0);
    synchronized (lastCustomer) {
      lastCustomer.notify();
    }
  }
}

/**
 * Write to stdout the name of the last customer.
 *
 */
public synchronized void showCustomerOut() {
  System.out.println("Customer " + lastCustomer.getName() + " is out");
}

static public void main(String[] args) {
  try {
    // read command-line arguments
    if (args.length != 2) {
      System.err.println("usage: BarberSemaphore nbChairs nbCustomers");
      System.exit(1);
    }
    int nbChairs = Integer.parseInt(args[0]);
    int nbCustomers = Integer.parseInt(args[1]);
    System.out.println("Starting BarberSemaphore with " + nbChairs
        + " chairs and " + nbCustomers + " customers");
```

```
      // create shop
      BarberMonitor shop = new BarberMonitor(nbChairs);
      // create barber
      new Barber(shop).start();
      // create customers
      for (int i = 0; i < nbCustomers; ++i) {
        new Customer(i, shop).start();
      }
    } catch (Exception e) { // report any exceptions
      System.err.println("Exception in BarberShop.main" + e);
    }
  }
}
```

1.3 Avec des Locks

Pour finir on réalisera une version du coiffeur dormeur utilisant l'interface *Lock* (utiliser la classe *R*eentrantLock). On créera un certain nombre de *C*ondition afin de réaliser la synchronisation entre les clients et le coiffeur.

Solution:

../java/BarberLock.java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

//===========================================================================
//BarberLock in Java
//Fabrice Peix --- ESSI --- 2006
//---------------------------------------------------------------------------
//Usage:
//javac BarberLock.java
//java BarberLock nbChairs nbCustomers
//===========================================================================

/**
 * Simulate Customer behaviors with Lock
 *
 * @author Fabrice Peix
 */
class CustomerLock extends Thread {
  static boolean notified = false;

  /** Waiting room capacity */
  private int nbChairs;

  /** Le Lock used to protect shared data */
  private Lock l;

  /** Used Condition */
  private Condition customerCond, barberCond, chairCond;
```

```
/**
 * @param numero
 * @param l
 *          The lock
 * @param customerCond
 *          Condition used by customer to waait
 * @param barberCond
 *          Condition used by barber to wait
 * @param barberCond
 *          Condition used by barber to call customer on his chair
 * @param nbChairs
 *          Waiting room capacity
 */
public CustomerLock(int numero, Lock l, Condition customerCond,
    Condition barberCond, Condition chairCond, int nbChairs) {
  super("Client " + numero);
  this.l = l;
  this.customerCond = customerCond;
  this.barberCond = barberCond;
  this.chairCond = chairCond;
  this.nbChairs = nbChairs;
}

/**
 * Customer go to barber
 *
 * @return true if haircut is done and false otherwise
 */
private boolean haircut() {
  l.lock();
  try {
    if (BarberLock.nbFreeChairs == 0) {
      System.out.println("Client " + getName() + " drink coffee...");
      return false;
    }
    if (BarberLock.nbFreeChairs == nbChairs) {
      System.out.println("Client " + getName() + " wake up barber");
      barberCond.signal();
    }
    BarberLock.nbFreeChairs--;
    while (!notified) {
      System.out.println("Client " + getName() + " wait for barber");
      customerCond.await();
    }
    notified = false;
    BarberLock.nbFreeChairs++;
    chairCond.signal();
    System.out.println("Client " + getName() + " is on barber chair");
  } catch (InterruptedException e) {
    e.printStackTrace();
  } finally {
    l.unlock();
```

```
    }
    return true;
  }

  public void run() {
    while (!haircut())
      try {
        sleep(2000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
  }
}

/**
 * Simulate barber behavior with Lock
 *
 * @author Fabrice Peix
 */
class BarberLock extends Thread {
  /** waiting room capacity */
  private int nbChairs;

  /** free chairs in barber waiting room */
  static public int nbFreeChairs;

  /** The Lock */
  private Lock l;

  /** Various Condition */
  private Condition customerCond, barberCond, chairCond;

  /**
   * Initialize barber
   *
   * @param l
   *          The lock
   * @param customerCond
   *          Condition used by customer to waait
   * @param barberCond
   *          Condition used by barber to wait
   * @param barberCond
   *          Condition used by barber to call customer on his chair
   * @param nbChairs
   *          Waiting room capacity
   */
  public BarberLock(Lock l, Condition customerCond, Condition barberCond,
      Condition chairCond, int nbChairs) {
    super();
    this.l = l;
    this.customerCond = customerCond;
    this.barberCond = barberCond;
    this.chairCond = chairCond;
```

```
  this.nbChairs = nbChairs;
  nbFreeChairs = nbChairs;
}

/*
 * (non-Javadoc)
 *
 * @see java.lang.Runnable#run()
 */
public void run() {
  while (true) {
    l.lock();
    try {
      while (nbFreeChairs == nbChairs) {
        System.out.println("Waiting for customers");
        barberCond.await();
      }
      CustomerLock.notified = true;
      customerCond.signal();
      chairCond.await();
      System.out.println("Begin hair cut");
      sleep(1000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      l.unlock();
    }
  }
}

public static void main(String args[]) {

  // read command-line arguments
  if (args.length != 2) {
    System.err.println("usage: BarberSemaphore nbChairs nbCustomers");
    System.exit(1);
  }
  int nbChairs = Integer.parseInt(args[0]);
  int nbCustomers = Integer.parseInt(args[1]);
  System.out.println("Starting BarberSemaphore with " + nbChairs
      + " chairs and " + nbCustomers + " customers");

  // Create Lock and Condition
  Lock realLock = new ReentrantLock();
  Condition customerCond = realLock.newCondition();
  Condition barberCond = realLock.newCondition();
  Condition chairCond = realLock.newCondition();

  // Create barber thread
  BarberLock barber = new BarberLock(realLock, customerCond, barberCond,
      chairCond, nbChairs);
  barber.start();
```

```
    // Create customers threads
    for (int i = 0; i < nbCustomers; i++) {
      new CustomerLock(i, realLock, customerCond, barberCond, chairCond,
          nbChairs).start();
    }
  }
}
```

2 Bonus : et s'il s'agissait d'une chaine de coiffure ?

Si vous avez eu le temps de faire travailler le coiffeur précédent à coups de moniteurs et de sémaphores, imaginons maintenant qu'il s'agisse d'une chaine de coiffure multinationale (Jean–Louis Dezange) qui souhaite passer à l'échelle en embauchant C coiffeurs.

Imaginons aussi que M. Dezange soit radin sur le nombre de ciseaux disponibles X (i.e. $X < C$, sachant qu'une paire de ciseaux de coiffeur coûte au minimum 150e), mais que ce n'est pas très grave car le tiers du temps, le client est en train de se faire laver les cheveux (et le shampoing coule à profusion chez M. Dezange). De toutes façons, après la coupe, il y a la phase de brushing où il n'y a pas besoin de ciseaux.

Pour que le salon tourne, M. Dezange a investi dans F fauteuils de coiffure et a aussi agrandi la salle d'attente. Mais M. Dezange conserve tout de même à l'esprit qu'un beau fauteuil en cuir, ça coûte, i.e. $F < C$ (NB : dans ce modèle, on ne compte pas les fauteuils pour laver les cheveux qui sont des banquettes où l'on peut tasser beaucoup de personnes ayant une corpulence raisonnable (D'ailleurs quand les coiffeurs dorment certains utilisent la banquette puisqu'il n'y a plus assez de fauteuils...)).

Vous êtes directeur des ressources humaines chez M. Dezange. Vous aimeriez tester sur un modèle de simulation le nombre de coiffeurs optimal C pour utiliser au mieux vos ressources "ciseaux, fauteuils" pour répondre à la demande des clients et pour minimiser le niveau sonore des ronflements de vos coiffeurs (on supposera que ce niveau sonore n'est pas dû aux ronflements des clients, la salle d'attente étant insonorisée).

Essayez d'adapter un de vos programmes (il y en a surement un pour lequel c'est plus simple) pour gérer plusieurs coiffeurs et la contrainte du nombre de ciseaux. Il est fort probable qu'il faille changer de classe pour l'acquisition des moniteurs, notamment si on suppose qu'il y a désormais F fauteuils de coiffure.

Pour que la simulation soit la plus réelle possible, essayez d'introduire un temps aléatoire pour le temps du shampoing, le temps de la coupe et le brushing.