Programmation avancée et répartie en Java : les accès concurrents

Frédéric Gava

L.A.C.L
Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique

Cours de M1 MIAGE
(d'après les notes de cours de Fabrice Mourlin)

Plan

1. Problématique et solution
2. Autres méthodes pour la concurrence

Plan

1. Problématique et solution
2. Autres méthodes pour la concurrence

Déroulement du cours

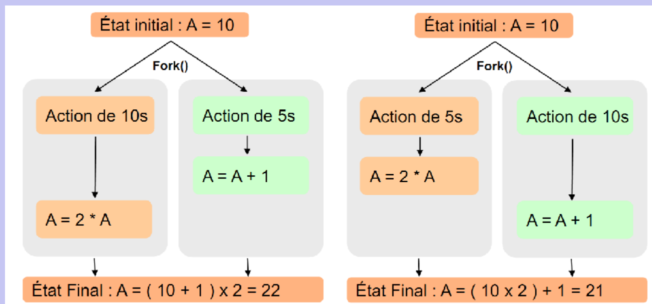
1. Problématique et solution
2. Autres méthodes pour la concurrence

Le problème des accès concurrent

data-race

Le problème est le suivant : quand 2 (ou plus) threads accèdent à une même zone mémoire (un objet, un tableau, une variable, *etc.*), ils impossible de savoir dans quel ordre se fera les lectures et surtout les écritures sur cette zone du fait de l'ordonnanceur.

Exemple simple

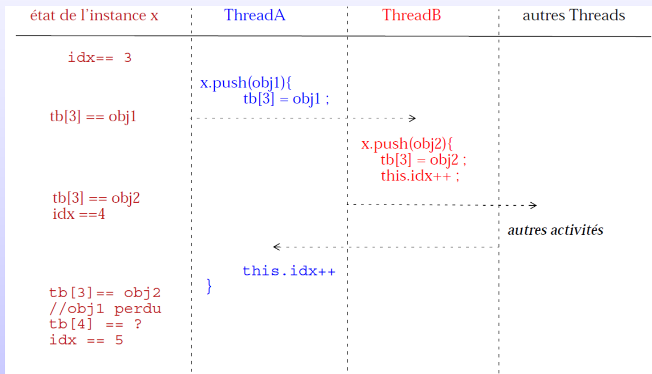


Exemple en Java (1)

```
class MaPile {

 ... // code divers dont définition de MAX
 private Object[] tb = new Object[MAX] ;
 private int idx = 0 ;

 public void push(Object obj) {
  ... //on suppose que lon traite le pb de la taille
  tb[idx] = obj ;
  this.idx++ ;
  }

 public Object pop() {
  ... // on suppose quon empêche de dépiler une pile vide (exception)
  this.idx-- ;
  return tb[idx] ;
  }

...// autre codes dont par ex. la "taille" de la pile
}
```

Et maintenant étudions ce qui se passerait si plusieurs threads pouvaient faire des demandes à la même instance de MaPile

Exemple en Java (2)



Le problème est que tb et idx (de l'instance) doivent être cohérent : si l'un est modifié, l'autre doit être modifié en conséquence. Lorsqu'un thread a commencé une exécution il faut garantir que le même thread sera seul autorisé à agir dans ce bloc. En Java, nous pouvons utiliser un mécanisme de moniteur (C.A.R. Hoare, 1971)

Les moniteurs

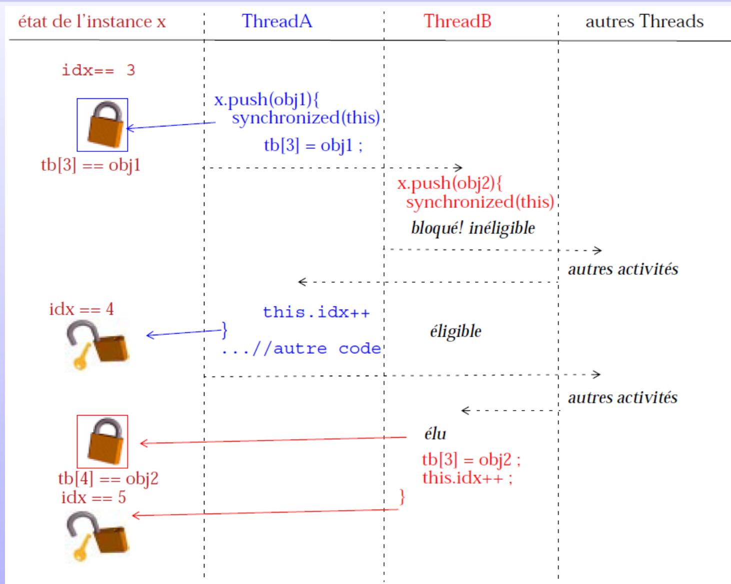
Bloc synchronized

```
public void push(Object obj) {
 ... //on suppose que l'on traite le pb de la taille
 synchronized(this)
     {tb[idx] = obj ; this.idx++ ; }
```

Conséquences

- un bloc **synchronized** permet de protéger une partie du code contre des accès concurrents
- En Java tout objet est doté d'un moniteur : c'est un dispositif qui gère un verrou et des files d'attente d'accès à ce verrou.
- Ici le bloc **synchronized** est rattaché à l'objet courant : tout thread qui va chercher à entrer dans le bloc devra tenter d'acquérir le verrou ou restera bloqué en file d'attente en attendant son tour d'acquérir le verrou

Comportement d'un moniteur



Les moniteurs (2)

Cohérence

Tous les codes manipulant une même valeur, qui ce doit de rester cohérente, doivent se synchroniser sur le même moniteur.

Donc pour compléter le code de la pile :

```
public Object pop() { ...; synchronized(this) {this.idx-- ;return tb[idx] ;} }
```

Les données du moniteur doivent être déclarées avec le mot clé **private** : seules les méthodes du moniteur accèdent à ces données.

Petitesse

Pour des raisons de performance, les blocs synchronized doivent être les plus étroits possible et se concentrer sur le code critique.

Toutefois, lorsque toute une méthode doit être synchronisée :

```
public synchronized void push(Object obj) {...}
```

Aussi, pour tester si le Thread courant a acquis le verrou sur un objet donné : `Thread.holdsLock(objet);`

Les moniteurs (3)

Synchro static

Une méthode de classe peut être **synchronized** : le moniteur est le champ statique .**class** de la classe courante (objet de type Class).

Dans un code **static** il est bien sûr possible de se synchroniser sur une variable partagée.

Tout objet

On peut aussi utiliser un objet passé en paramètre d'une méthode.

```
static Comparable[] trier(Comparable[] tb) {
  synchronized(tb) {
      // suppose que tout utilisateur de ce tableau
      // soit au courant! et fait lui même appel
      // au moniteur avant de re−arranger les éléments
      ...
      return tb; // sort du bloc synchronized
```

Sinon il y a risque de :

Interblocage (1)

Une étreintes mortelles (*deadlock*)

Quand un thread bloque l'accès à une resource par un verrou en attendant qu'un autre thread, lui-même bloqué par un verrou, libère une ressource ou calcul une valeur adéquate.

Exemple : Si le code exécuté par un Thread A acquiert un moniteur X, puis tente d'acquérir un moniteur Y. Mais si un Thread B a acquis Y et tente d'acquérir X, tout est bloqué !

Pour un moniteur partagé entre plusieurs classes (passé en paramètre p. ex.) le risque existe.

Java ne détecte pas automatiquement les deadlocks.

Synchronised

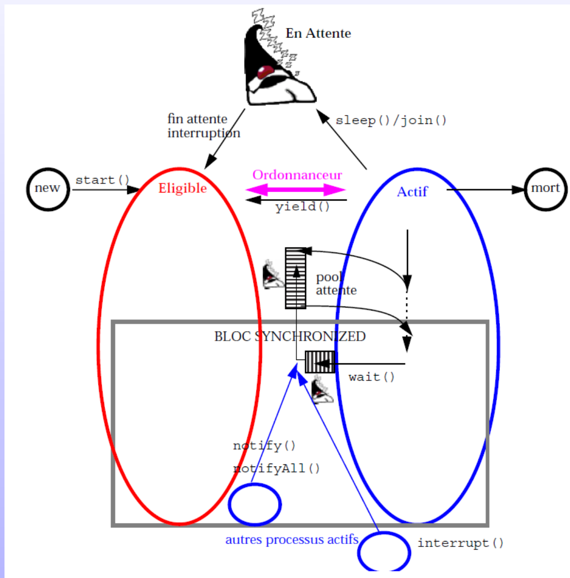
Rôle

Utilisé dans la programmation multithread dans la déclaration d'une méthode ou d'un bloc d'instructions pour indiquer que seul un thread peut accéder en même temps à ce bloc ou à cette méthode

4 manières d'utilisation

```
(1) public void synchronized methode(){
      print("2 threads ne peuvent pas appeler cette méthode 'en même temps'");
      ...
(2) public void methode(){
       synchronized(this){
          print("2 threads ne peuvent pas exécuter ce code 'en même temps'");
       ...
(3) synchronized(monObjet) {... // code}
(4) public static void methode() {synchronized(MaClass.class) {... // code} ... }
```

État de vie d'un thread bis



Compléments

Constructeurs

Le code d'un constructeur ne peut pas être synchronized.

Il n'est exécuté que par un seul thread pour la création d'un objet donné et, normalement, une référence sur l'objet en construction ne devrait pas être rendue avant la fin de l'exécution du constructeur. Le danger vient de l'emploi dans le constructeur de méthodes qui cachent le fait qu'elles passent **this** à d'autres objets.

Méthodes deprecated

Les méthodes suspend() (pour suspendre l'activité), resume() (pour faire repartir un thread suspendu) et stop() pour l'arrêter.

Ces méthodes permettent à un thread d'agir sur un autre thread. Obsolètes pour des raisons de sécurité. La méthode interrupt reste intéressante pour réveiller des threads abusivement somnolents.

Tubes entre threads JAVA

Java fournit des classes permettant de créer simplement des tubes de communication entre threads. On manipule un tube à l'aide de deux classes : PipedWriter pour l'entrée du tube (où l'on écrira des données), et PipedReader pour la sortie du tube. Exemple :

```
PipedWriter out = new PipedWriter();
PipedReader in;
try {in = new PipedReader(out);} catch (IOException e) { ... }
```

PipedWriter est une sous-classe de Writer, et possède donc une méthode write que l'on appelle pour écrire des données dans le tube. PipedReader, sous-classe de Reader, a diverses méthodes read permettant de lire un ou plusieurs caractères depuis le tube.

Le programme crée ensuite des threads en leur donnant l'extrémité du tube (l'objet in ou out) dont elles ont besoin.

La problèmatique des producteurs-consommateurs

Le pattern

Un ou des producteurs ajoutent des objets dans une file (FIFO).
Un ou des consommateurs enlèvent ses objets de la file.

Exemple : Un magasin (producteur) et des paniers.

Implémentation naive

```
public class FIFO {
 private ArrayList liste = new ArrayList();
 public Object prendre() {return liste.get(0) ;}
 public void poser(Object obj) { liste.add(obj) ;}
 }
```

Partageons cette classe entre plusieurs threads.

Le problème de partage des consommateurs

Plusieurs consommateurs

Si la file est vide, d'une part, la méthode get propage une exception (qu'il va falloir traiter) et, d'autre part, il va falloir attendre qu'un producteur veuille bien remettre un objet dans la file.

Une boucle active (tester la taille de la liste) entraînerait un polling

Description de la solution

Il faut un mécanisme tel que le consommateur "affamé" (face à une file vide) puisse passer dans un état d'inéligibilité qui durera tant qu'un producteur n'a pas signalé qu'il a déposé un objet.

Le mécanisme du moniteur va être mis à contribution : un objet va servir de support à un dispositif de coordination entre threads.

Mise en oeuvre

```
public class FIFO {
 private ArrayList liste = new ArrayList();
 public synchronized Object prendre() {
  while( 0 == liste.size()) {
   try {this.wait();}
    catch (InterruptedException exc) {// on peut toujours logger et gérer les exeptions}
  }
  return liste.get(0) ;
 }
 public synchronized void poser(Object obj) {this.notify(); liste.add(obj);}
```

Le principe est le suivant :

- si la liste est vide alors les threads affamés vont se mettre dans un pool associé à l'instance courante. La méthode wait() met le thread qui l'exécute dans ce pool.
- un thread qui vient de déposer un objet signale que les threads en attente peuvent redevenir éligibles. Il exécute sur l'objet support du moniteur la méthode notify()

Mise en oeuvre : remarques (1)

- Les méthodes wait et notify (de Object) ne peuvent être appelées que si le thread courant a acquis le moniteur sur l'objet concerné. Cela n'est vérifié qu'runtime et peut génèrer une IllegalMonitorStateException ("thread not owner"). Le bloc **synchronized** englobe tout le code "fragile" (être sûr au moment du get(0) que la liste n'a pas été modifiée/vidée). notify avant le add n'est donc pas incorrect.
- wait() peut etre interrompu (voir méthode interrupt()) et exige donc d'être placé dans un bloc try-catch.
- Le thread qui exécute wait() est mis dans un *pool* d'attente ; quand il en sort il "revient" dans le bloc **synchronized** et entre en compétition avec d'autres threads pour acquérir le verrou. Il n'est donc pas certain que la condition du notify soit encore remplie au moment où le thread acquiert enfin le verrou : le test sous forme while est obligatoire.

Mise en oeuvre : remarques (2)

- notify existe sous deux formes :
  1. notify(), un thread est pris dans le wait-pool
  2. notifyAll(), tous les threads du wait-pool vont chercher à acquérir le verrou.

  Noter que le choix d'un Thread particulier dans le pool est dépendant d'une stratégie de réalisation de la JVM : le programmeur ne doit donc pas faire d'hypothèses à ce sujet.
- Invoquer notify() ou notifyAll() alors qu'il n'y a aucun thread dans le wait-pool ne prête pas à conséquence, on peut donc le faire sans risque.
- Une version de wait permet de limiter le temps d'attente : wait(**long** Millisecondes). Bien entendu, à la "sortie", on n'est vraiment pas sûr que la condition attendue soit remplie.

Deuxième mise en oeuvre (1)

```
interface ProdConsInterface {
  public void Deposer(int m) throws InterruptedException ;
  public int Prelever() throws InterruptedException ;
}

class ProdConsMonitor implements ProdConsInterface {
  private int NbPleins=0, tete=0, queue=0 ;
  private int N ;
  private int tampon[];
  ProdConsMonitor (int n) {N=n ;tampon = new int [N] ;
  }

public synchronized void Deposer (int m) throws InterruptedException {
  while (NbPleins == N) { wait(); }
  tampon[queue] =m ;
  queue = (queue +1)% N;
  NbPleins ++;
  System.out.println(Thread.currentThread().getName() +" vient de produire " + m) ;
  notifyAll();
  }
```

Deuxième mise en oeuvre (2)

```
public synchronized int Prelever () throws InterruptedException {
  while (NbPleins == 0 ) {wait(); }
  int m= tampon[tete];
  tete = (tete + 1)% N;
  NbPleins --;
  notifyAll();
  System.out.println(Thread.currentThread().getName() + " vient de consommer " + m) ;
  return m ; }
} // fin de ProdConsMonitor

class Producteur extends Thread {
  ProdConsMonitor Mo;
  Producteur ( ProdConsMonitor Mo ) {this.Mo=Mo;}
  public void run() {
   try {
    while (true) {
      int m= (int)(1000*Math.random());
      Mo.Deposer(m) ;
      Thread.sleep((int)(1000*Math.random()));
    } } catch(InterruptedException e) {}
  }// fin run
} // fin de la classe Producteur
```

Deuxième mise en oeuvre (3)

```
class Consommateur extends Thread {
 ProdConsMonitor Mo;
 Consommateur (ProdConsMonitor Mo) {this.Mo=Mo;}
 public void run() {
  try {
    while (true) {
      int m = Mo.Prelever() ;
      Thread.sleep((int)(1000*Math.random()));
    }
  } catch(InterruptedException e) {}
 } // fin run
} // fin de la classe Consommateur

class TheMain {
 public static void main(String argv[]) {
  int N= Integer.parseInt(argv[0]);
  ProdConsMonitor pc = new ProdConsMonitor(N) ;
  Producteur producteur = new Producteur (pc) ;
  Consommateur consommateur = new Consommateur (pc) ;
  producteur.setName("Producteur") ; producteur.start() ;
  consommateur.setName("Consommateur") ; consommateur.start() ;
  producteur.join(); consommateur.join();
```

Déroulement du cours

1. Problématique et solution

2. Autres méthodes pour la concurrence

Inconvénients des moniteurs

Inconvénients (codage difficile)

- On ne peut pas bloquer une thread sur une condition particulière ni en réveiller un car toutes les threads bloquées sont dans la file d'accès au moniteur
- Pouvoir itérer avec des threads sur des collections de données

Solutions (>1.5)

1. Verrous (lock), Sémaphores, Futures, *etc.*
2. Diverses collections concurrentes

Nous allons maintenant décrire (non-exhaustif) ces solutions.

Tâches avec résultats

Définition, Callable et Future

Équivalent de Runnable mais permettant de renvoyer une valeur.

Une seule méthode pour l'interface générique Callable<V> :

```
V call() throws Exception
```

Utilisation

Soumettre à un ExecutorService

```
Future<V> submit(Callable<V> tache)
```

Soumettre la tâche à l'exécuteur, le Future<V> renvoyé permet de manipuler de manière asynchrone la tâche.

Tâches en attente

Implémentée dans FutureTask<V> :

- V get() : renvoie la valeur du résultat associée à la tâche en bloquant jusqu'à ce que celui-ci soit disponible,
- **boolean** isDone() : renvoie true si la tâche est terminée.
- **boolean** cancel(**boolean** meme_en_Cours) : annule la tâche si celle n'a pas encore commencé. Si meme_en_cours, tente de l'arrêter aussi même si elle a déjà débuté,
- **boolean** isCancelled() : renvoie **true** si la tâche a été annulée.

Exemple avec Futures

```
class ImageRenderer { Image render(byte[] raw); }
class App { // ...
 Executor executor = ...; // any executor
 ImageRenderer renderer = new ImageRenderer();
 public void display(final byte[] rawimage) {
  try {
   Future<Image> image = Executors.invoke(executor, new Callable<Image>(){
       public Image call() {
       return renderer.render(rawImage);
       }});
   drawBorders(); // do other things ...
   drawCaption(); // ... while executing
   drawImage(image.get()); // use future
  }
  catch (Exception ex) {cleanup();return;}
 }
}
```

Les sémaphores (Java 1.5 et plus)

Définition

Un sémaphore est un objet de haut niveau permettant de gérer l'accès concurrent à une ressource partagée, qui accepte au plus n accès concurrents.

Utilisation

```
Semaphore s = new Semaphore (1, true);
s.acquire();
//Accès (code) à la section critique;
s.release();
```

Les threads peuvent être débloqués par une InterruptedException :

```
try {
 sem.acquire ();
 // Utilisation de la ressource protégée par sem
 } catch ( InterruptedException e){...}
finally { sem.release ();// toujours éxecutée }
```

Les sémaphores (2)

Paquetage java.util.concurrent avec :

public class Semaphore **extends** Object **implements** Serializable

Constructeur :

- Semaphore(**int** permits) : Crée un Sémaphore avec une valeur initiale (nb d'appels non bloquants)
- Semaphore(**int** permits, **boolean** fair) : Crée un Sémaphore avec une valeur initiale (nb d'appels non bloquants). fair=true pour garantir une gestion FIFO des processus en cas de conflit.

Quelques Méthodes :

- **void** acquire () : demande une permission (un ticket), le thread appelant se bloque si pas de ticket ou si est interrompu.
- **void** acquire (**int** permits) : demande un certain nombre de tickets, la thread appelante se bloque si pas de ticket ou bien si elle est interrompue.
- **void** release () : libère un ticket.
- **void** release (**int** permits) : libère des tickets.

Exemple avec sémaphore (1)

```
import java.util.concurrent.*;

interface ProdConsInterface {
  public void Deposer(int m) throws InterruptedException ;
  public int Prelever() throws InterruptedException ;
}

class ProdConsSem implements ProdConsInterface {
  Semaphore SVide, SPlein;
  int tampon[];
  int queue,tete, N;
  ProdConsSem (int taille) {
    N=taille ;
    tampon = new int [N] ;
    SVide=new Semaphore (N, true);
    SPlein=new Semaphore(0,true);
  }
```

Exemple avec sémaphore (2)

```
    public void Deposer (int m) throws InterruptedException {
      SVide.acquire();
      tampon[queue] =m ;
      queue = (queue +1)% N;
      System.out.println(Thread.currentThread().getName() +" vient de produire " + m) ;
      SPlein.release();
    }

    public int Prelever () throws InterruptedException {
      SPlein.acquire();
      int m= tampon[tete];
      tete = (tete + 1)% N;
      System.out.println(Thread.currentThread().getName() +" vient de consommer " + m) ;
      SVide.release();
      return m ;
    }
} // fin de ProdConsSem
```

Exemple avec sémaphore (3)

```
class Producteur extends Thread {
  ProdConsSem Sem;
  Producteur ( ProdConsSem Sem ) {this.Sem=Sem;}
  public void run() {
   try {
    while (true) {
      int m= (int)(1000*Math.random());
      Sem.Deposer(m) ;
      Thread.sleep((int)(1000*Math.random()));
    }
   } catch(InterruptedException e) {}
  }// fin run } // fin de la classe Producteur

class Consommateur extends Thread {
  ProdConsSem Sem;
  Consommateur (ProdConsSem Sem) {this.Sem=Sem;}
  public void run() {
   try {
    while (true) {
      int m = Sem.Prelever() ;
      Thread.sleep((int)(1000*Math.random()));
    }
   } catch(InterruptedException e) {}
  } // fin run } // fin de la classe Consommateur
```

Exemple avec sémaphore (4)

```
//La classe main : à exécuter avec un argument = taille du tampon
class ProdConsMain {
 public static void main(String argv[]) {
   int N;
   try{ N=Integer.parseInt(argv[0] );}
    catch(ArrayIndexOutOfBoundsException exp) {
      System.out.println("USAGE : java ProdConsMain [taille du tampon]");
      System.exit(0);
      }
   ProdConsSem pc = new ProdConsSem(N) ;
   Producteur producteur = new Producteur (pc) ;
   Consommateur consommateur = new Consommateur (pc) ;
   producteur.setName("Producteur") ;
   producteur.start() ;
   consommateur.setName("Consommateur") ;
   consommateur.start() ;
  }
}
```

Autres primitives des sémaphores

Par tentative : l'appel n'est plus bloquant et la valeur retournée est true si l'accès est possible :

- **boolean** tryAcquire()
- **boolean** tryAcquire(**int** tickets)
- **boolean** tryAcquire(**long** timeout, TimeUnit unit)
- **boolean** tryAcquire(**int** tickets, **long** timeout, TimeUnit unit) ;
  Attention : Interruption possible par InterruptedException.

Acquisition sans pouvoir être interrompus :

- acquireUninterruptibly()
- acquireUninterruptibly(**int** tickets)

Opérations Particulières

Il s'agit d'opérations combinant, de manière atomique, un test et une affectation éventuelle (ici, dans le cas d'un AtomicLong, une variable atomiqe) :

- **boolean** compareAndSet(**long** expect, **long** update) : teste l'égalité de la valeur avec expect et affecte update dans le cas positif. Revient à lire puis écrire un volatile
- **boolean** weakCompareAndSet(**long** expect,**long** update) : idem, sauf que l'accès n'est séquentialisé que sur cet objet
- **long** getAndSet(**long** newValue) : affecte à la nouvelle valeur et renvoie l'ancienne
- **long** getAndIncrement(), **long** getAndDecrement(), **long** incrementAndGet(), **long** decrementAndGet(), *etc.*

Comme pour toutes les variables atomiqes, ces opérations utilisent des instructions de très bas niveauen C/assembler et sont donc sujettes à variations selon les plateformes (et notamment ne sont pas, à strictement parler, non bloquantes).

Les variables à condition en Java 1.5 (et plus)

Paquetage java.util.concurrent.locks avec :

public interface Condition

Remplace l'utilisation des méthodes du moniteur. Méthodes :

- **void** await() : provoque le blocage du thread appelant jusqu'à réception d'un signal ou bien d'une interruption.
- **void** await (**long** time, TimeUnit unit) : provoque le blocage du thread appelant jusqu'à réception d'un signal, d'une interruption ou bien le temps unit est dépassé.
- **void** signal() : réveille un thread bloqué.
- **void** signalAll() : réveille toutes les threads bloqués.

Les verrous en Java 1.5 (et plus)

Paquetage java.util.concurrent.locks avec :

public interface Lock

Remplace l'utilisation des méthodes synchronized (accessibles en exclusion mutuelle). Méthodes :

- **void** lock() : le thread appelant obtient un verrou.
- **void** lockInterruptibly() : le thread appelant obtient un verrou et le garde jusqu'à son interruption.
- Condition newCondition() : retourne une nouvelle instance Condition à utiliser avec un verrou de type Lock.
- **void** unlock() : restitue le verrou.

Les verrous (2)

Paquetage java.util.concurrent.locks avec :

public class ReentrantLock **extends** Object **implements** Lock , Serializable

Un verrou d'exclusion mutuelle. Méthodes :

- **void** lock() : le thread appelant obtient un verrou.
- **void** lockInterruptibly() : le thread appelant obtient un verrou et le garde jusqu'à son interruption.
- ConditionObject newCondition() : retourne une nouvelle variable Condition liée à une instance de Lock.
- **void** unlock() : restitue le verrou.

Exemple

```
class X {
 private final ReentrantLock verrou = new ReentrantLock();
 // ...
 public void m() {
   verrou.lock(); // bloque jusqu'à satisfaction de la condition
   try { // ... Corps de la méthode
       }
   finally {verrou.unlock() }
```

Exemple d'un buffer

```
class BoundedBuffer {
  Lock lock = new ReentrantLock();
  Condition notFull = lock.newCondition();
  Condition notEmpty = lock.newCondition();
  Object[] items = new Object[100];
  int putptr, takeptr, count;
  public void put(Object x)throws IE {
    lock.lock(); try {while (count == items.length)
                        notFull.await();
                        items[putptr] = x;
                        if (++putptr == items.length) putptr = 0;
                        ++count;
                        notEmpty.signal();
                      } finally {lock.unlock();}
  }
  public Object take() throws IE {
   lock.lock(); try { while (count == 0) notEmpty.await();
                        Object x = items[takeptr];
                        if (++takeptr == items.length) takeptr = 0;
                        --count;
                        notFull.signal();
                        return x;
                      } finally { lock.unlock(); }
```

Exemple du producteur/consommateur (1)

```
import java.util.concurrent.locks.*;

interface ProdConsInterface {
  public void Deposer(int m) throws InterruptedException ;
  public int Prelever() throws InterruptedException ;
}

class ProdConsMonitor implements ProdConsInterface {
  final Lock verrou = new ReentrantLock();
  final Condition Plein = verrou.newCondition();
  final Condition Vide = verrou.newCondition();
  int tampon[];
  int queue,tete, N, NbPleins=0,NbVides=0;
  ProdConsMonitor (int n) {
    N=n ;
    tampon = new int [N] ;
  }
```

Exemple du producteur/consommateur (2)

```
public void Deposer (int m) throws InterruptedException {
  verrou.lock();
  try {
    if (NbPleins == N) Plein.await();
    tampon[queue] =m ;
    queue = (queue +1)% N;
    NbPleins ++;
    System.out.println(Thread.currentThread().getName() +" vient de produire " + m) ;
    Vide.signal();
   } finally {verrou.unlock();}
}
public int Prelever () throws InterruptedException {
  verrou.lock();
  try {
   if (NbPleins == 0 ) Vide.await();
   int m= tampon[tete];
   tete = (tete + 1)% N;
   NbPleins --;
   Plein.signal();
   System.out.println(Thread.currentThread().getName() +" vient de consommer " + m) ;
   return m ;
  } finally {verrou.unlock();
}
```

Exemple du producteur/consommateur (3)

```
class Producteur extends Thread {
  ProdConsMonitor Mo;
  Producteur (ProdConsMonitor Mo ) {this.Mo=Mo;}
  public void run() {
   try {
    while (true) {
     int m= (int)(1000*Math.random());
     Mo.Deposer(m) ;
     Thread.sleep((int)(1000*Math.random()));
   }
  } catch(InterruptedException e) {} } }

class Consommateur extends Thread {
  ProdConsMonitor Mo;
  Consommateur (ProdConsMonitor Mo) {this.Mo=Mo;}
  public void run() {
    try {
      while (true) {
       int m = Mo.Prelever() ;
       Thread.sleep((int)(1000*Math.random()));
      }
   } catch(InterruptedException e) {} } }
```

Faire des barrières (1)

Définition

Une barriére est un système permettant de bloquer un thread en un point jusqu'à ce qu'un nombre prédéfini de threads atteigne également ce point.

La barrière est cyclique car elle est réutilisable. Pour une barrière utilisée une seule fois, préférer un CountDownLatch. Sinon :

- CyclicBarrier(**int** n) : barrière pour n threads,
- CyclicBarrier(**int** n, Runnable barrierAction) : barrière où barrierAction est exécutée avant de lever la barrière.
- **int** await() : attendre que tous les threads aient atteint la barrière ou
  - que l'un des threads en attente soit interrompu
  - l'un des threads en attente atteigne son délai d'attente
  - la barriere soit réinitialisée

  La valeur de retour : ordre inverse d'arrivée à la barriére, c'.-à-d. le nombre de threads restant à attendre.

Faire des barrières (2)

- **int** await(**long** timeout, TimeUnit unite) : idem avec un délai
- **int** getNumberWaiting() : renvoie le nombre de threads en attente sur la barrière (pour débug)
- **void** reset() : réinitialise la barrière, i.e. les éventuels threads en attente reçoivent BrokenBarrierException. Si une barrière est interrompue autrement que par reset(), il est difficile de la réinitialiser correctement. Dans ce cas, il est conseillé de créer une nouvelle barrière.

Exemple barrière :

```
// une seule fois
CyclicBarrier barriere = new CyclicBarrier (N);
...
// pour chacun des N threads
try {barriere.await();}
 catch (InterruptedException ex) {return ;}
 catch (BrokenBarrierException ex) {return;}
// réinitialisation de la barrière
barriere.reset();
```

Faire des barrières (3)

Avec CountDownLatch : Attente sur des évènements (compte à rebours) et non plus sur l'arrivée d'autres threads. Méthodes :

- CountDownLatch(**int** n) : crée une barrière de n éléments,
- **void** await() : attendre que le compte à rebours soit terminé,
- **void** await(**long** delai, TimeUnit unit) : avec delai,
- **void** countdown() : décrémente le compteur. Si 0 est atteint, tous les threads bloqués sont libérés,
- **long** getCount() : renvoie l'état du compteur.

Les structures de données partagées (1)

Définition

Une collection (structure de donnée) est partagée si elle peut être accédée par plusieurs activités.

Difficultés (pour avoir une interface similaire) :

- Une collection ne peut, en général, pas se trouver dans un état arbitraire. P. ex, la liste chaînée.
- Pouvoir itérer en parallèle sur les données de la collection
- Pas bloquer tous les threads pour les écritures (zone critique)

Solutions

- Le confinement consiste à ne laisser accessible un objet que par un seul thread. P. ex. l'interface graphique utilise un seul thread pour gérer les évènements graphiques.
- Un objet immutable (**final**) est toujours *thread-safe*.
- Les collections concurrentes ! ! !

Collections Non Synchronizées

Protection avec :

- **synchronized**(objet)
- un Wrapper comme Collection.synchronizedMap(**new** HashMap<**int**,String>())

Ou bien des collections de base *thread-safe*

- **interface** List<T> : ArrayList<E>, LinkedList<E>, CopyOnWriteArrayList<E> (modifications coûteuses)
- **interface** Set<E> : HashSet<E>, TreeSet<E>, CopyOnWriteArraySet<E> (idem)
- **interface** Map<K,V> : table d'association clef-valeur. HashMap<K,V>, TreeMap<K,V> (arbre rouge-noir)
- Et attention si les éléments sont mutables...

Collections Synchronisées (1)

L'implémentation de ces objets utilisent des verrous (implicites).
Exemple des Vectors<E> (tableaux dynamiques) :

- **boolean** add(E elt) : ajoute elt en fin de tableau,
- **boolean** add(**int** i,E elt) : ajoute elt en position i
- E get(**int** index) : renvoie l'élément en position i du tableau
- **boolean** remove(E elt) : retire la première occurence de elt et retourne vrai s'il y en a une
- E remove(**int** i) : retire l'élément en position i et retourne celui-ci.

Collections Synchronisées (2)

Exemple des piles Stack<E> **extends** Vector<E> :

- E peek() : renvoie le premier élément de la pile sans l'enlever,
- E pop() : renvoie le premier élément de la pile et l'enlève de celle-ci,
- E push(E elt) : met elt sur la pile et retourne elt

Exemple des tables de hachage Hashtable<K,V> :

- V get(Object clef) : renvoie l'objet indexé par clef,
- V put(K clef, V valeur) : associer valeur à clef dans la table.

Itérateurs et Accès Concurrents

Il est possible d'utiliser des "itérateurs" :

```
Vector <E> liste ;
...
for (E elt : liste ) { traitement_sur elt;};
```

Problèmes :

- le comportement de l'itérateur est non-spécifié si le "traîtement" modifie la collection itérée
- Les objets/méthodes précèdentes sont défaillantes si une modification simultanée est détectée lors de l'utilisation d'un itérateur (exception ConcurrentModificationException).

Il est donc à la fois pénible et source d'erreurs de recoder des primitives de synchronisation de haut niveau. Utilisons les collections spéciales de java.util.concurrent :

- Collections spécialisées, p. ex. BlockingQueue
- Objets threadsafe mais sans verrous
- Implémentation efficace

Interface BlockingQueue (1)

Définition

Implémentations de files d'attente bloquantes (i.e. les accès bloquent si la file est vide ou pleine)

Rappel : bloquant ⇒ **throws** InterruptedException.

ArrayBlockingQueue<E> file de taille bornée. Méthodes :

- ArrayBlockingQueue(**int** n) construit une file de taille n
- ArrayBlockingQueue(**int** n,**boolean** fair) idem et équitable, les accès sont FIFO,
- **void** put(E elt) ajoute elt en attendant si la file est pleine,
- E take() renvoie le premier élément de la file et le retire, en bloquant tant que la file est vide,
- E poll(**long** delai, TimeUnit unit) renvoie le premier élément de la file et le retire, en attendant au plus delai unit,
- offer(E elt) ajoute elt à la file, en retournant immédiatement sans ajouter si la file est pleine.

Interface BlockingQueue (2)

Objets n'acceptant pas l'élément null...

- LinkedBlockingQueue<E> file optionnellement bornée :
  - LinkedBlockingQueue() crée une file d'attente (de taille maximale Integer.MAX_VALUE),
  - LinkedBlockingQueue(**int** n), comme précédèment
- SynchronousQueue<E> file de capacité nulle ⇒ "rendez-vous" :
  - SynchronousQueue()
  - SynchronousQueue(**boolean** fair), avec accès FIFO si fair vrai
  - même méthodes (certaines sont trivialisées).
- ConcurrentLinkedQueue<E>, Une file d'attente non bornée, "thread-safe" et sans attente :
  - **boolean** add(E elt) ajoute et renvoie true,
  - E poll() retourne et enlève le premier éléement de la file,
  - E peek() retourne sans enlever le premier élément
  - size() est de complexité O(n)...

ConcurrentHashMap<K,V>

- Table de hachage pour accès concurrents
- “thread-safe” et sans verrous (bon débit),
- impossible de verrouiller (synchronized) l’accès à toute la table,
- pas de ConcurrentModificationException avec les itérateurs même si ceux-ci sont prévus pour être mono-threads
- ConcurrentHashMap(**int** capInitiale,**float** facteurCharge, **int** niveauConcurrence) crée une table de capacité initiale capInitiale, avec facteurCharge de paramètre de rééquilibrage interne dynamique, et pour environ niveauConcurrence threads modifiant la table simultanément. Les rééquilibrages internes sont coûteux.
- Remplace très avantageusement Hashtable

Que Choisir ?

- Objets synchronizés (bloquants ou potentiellement bloquants) vs concurrents (non bloquants) ? Prendre en compte :
  - débit de traitement
  - taux de concurrence
  - type de traitement
  - intensif en E/S ou en CPU ?
  - prévisibilité de comportement
- Sémaphores, verrous, moniteurs, TestAndSwap ? ⇒ suivant les besoins et votre compréhension des méthodes des objets. Mieux vaut un peu lent mais sans inter-blocage !
- Future, Executor ⇒ proche de la programmation distribuée. Confusion possible.

Bien lire les documentations (JavaDoc ! ! ! !) et bien modéliser le problème (combien de threads modifient la structure, combien ne font que lire, problèmes de débit, *etc.*)

Au travail !