(Archi4)
Architecture 4
Programmation répartie
Accès concurrentiels

Lecture *Architecture 4*
Programmation répartie 17 janvier 2020

Miklós MOLNÁR

contacter
molnar@lirmm.fr
Institut Universitaire de Technologie de Montpellier

Accès concurrents

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Dans un système ou dans le réseau, il y a de nombreux processus qui veulent accéder aux ressources (CPU, imprimante, BD, service réseau, etc.)

Il faut définir des mécansimes pour

- organiser (ordonnancer) l'utilisation des ressources,
- assurer l'accès unique à une ressource qui ne peut pas être partagée,
- synchroniser le fonctionnement,
- assurer la communication entre eux, ...

Ici, on résume les problèmes et des mécanismes de base.

Accès concurrents

Exemples et terminologie

- Train sur une voie unique

  - Trains : des ”processus” ou ”des tâches”
  - Voie unique : *section critique*
  - Contrôle de la voie : *exclusion mutuelle* - méthode qui permet de s’assurer que si un processus ou thread utilise une ressource (partagée), les autres processus seront excluent de cette même activité

Une solution : verrouiller la voie par des sémaphores

- L’ exclusion mutuelle empèche que deux trains (threads) se retrouvent simultanément sur la voie unique (dans leurs sections critiques)
- La section critique d’un programme doit être une opération indivisible, atomique vu de l’extérieur

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Exemples et terminologie

Est-ce un verrouillage simple est toujours bon ?

- Exemple plus complexe - Dîner des philosophes
  - Un problème d'utilisation concurrentielle des ressources

  - Philosophe (un thread)
    - il réfléchit
    - il peut avoir faim (il attrape les fourchettes libres)
    - il mange s'il a deux fourchettes
  - Pour organiser le diner : ordonnancement, sémaphores, ...
    - Malgré les sémaphores, une mauvaise allocation de fourchettes conduit à un interblocage
    - Plusieurs solutions sont envisageables :
      - un ordonnanceur externe donne les droits
      - négociations entre les philosophes
      - temps d'attente (sleep) aléatoire

Architecture 4
Programmation réparti

Miklós MOLNÁR

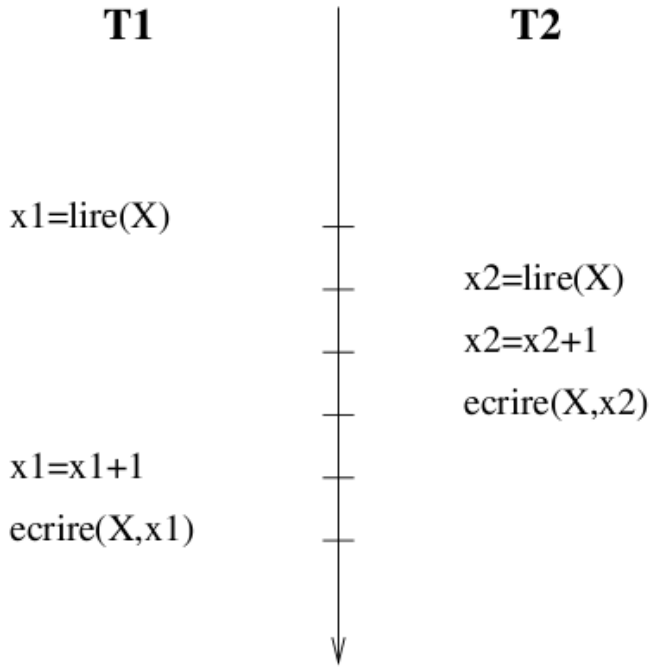
contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Problèmes liés à l'utilisation concurrentielle des variables

X est une variable commune des tâches *T*1 et *T*2

- Perte de mises à jour
  - une tâche T1 exécute une mise a jour calculée à partir d'une valeur lue (périmée)
  - la valeur lue est modifiée (depuis la lecture de T1) par T2
  - la mise à jour de T2 est donc écrasée



Miklós MOLNÁR

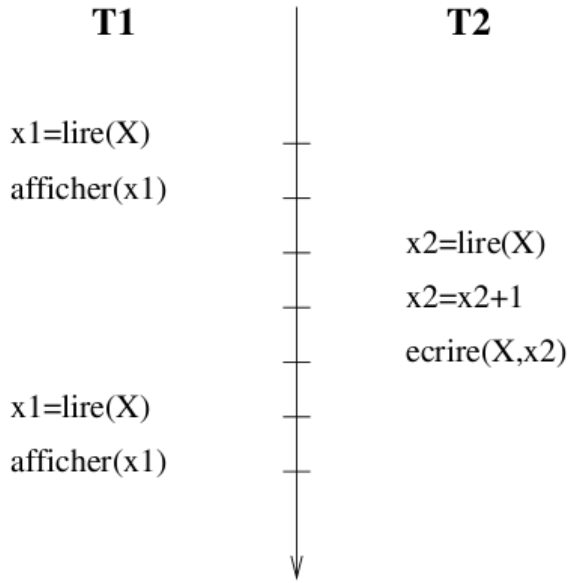
contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Problèmes liés à l'utilisation concurrentielle des variables

X est une variable commune des tâches *T*1 et *T*2

- Lecture non reproductible
  - une tâche relit une valeur déjà lue par elle mais obtient une valeur différente
  - la valeur a été modifiée par une autre tâche entre temps



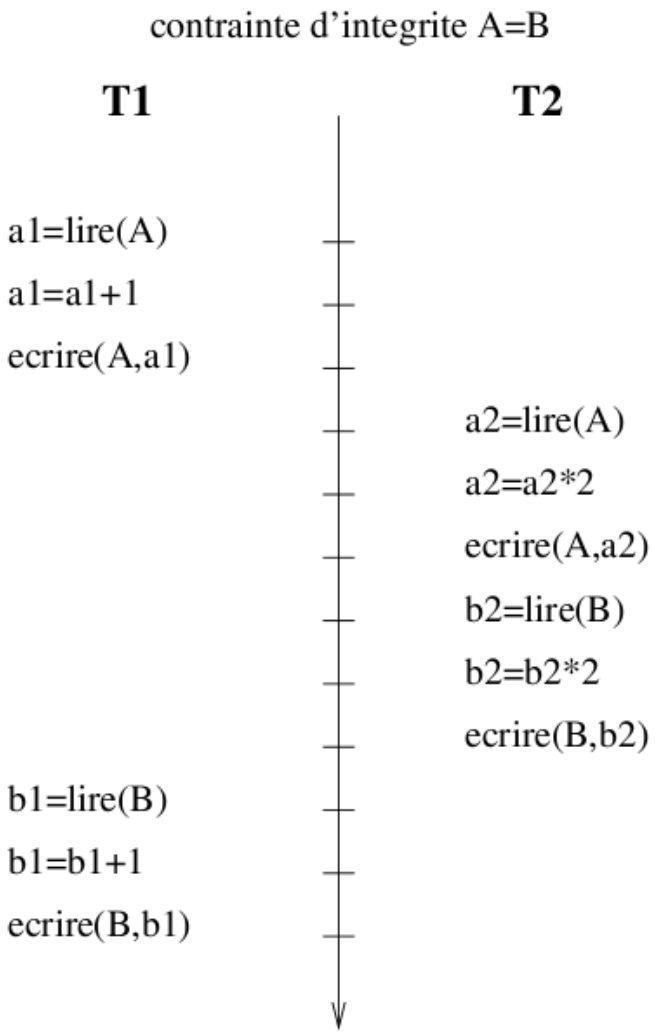
Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Problèmes liés à l'utilisation concurrentielle des variables

- Incohérence de mises à jour
  - des valeurs liées par une contrainte d'intégrité (ici A et B) sont mises à jour par 2 tâches dans des ordres différents
  - ce qui implique la violation de la contrainte



Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Problèmes liés à l'utilisation concurrentielle des variables

Une première idée : organiser une section critique avec un verrou pour éviter la perte et les incohérences

- Un verrou est un objet qui possède une clé qu'il faut avoir pour permettre une activité
- Poser un verrou sur la section (instructions) critique réalise une exclusion mutuelle :

```
debut
    poser un verrou
    x = lire(X)
    x = x + 1
    ecrire(X,x)
    liberer le verrou
 fin
```

- Cette section indivisible peut être exécutée par une seule tâche à la fois

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Problèmes liés à l'utilisation concurrentielle des variables

Une solution plus intéressante : verrouillage deux phases

- Implémentation du verrouillage :
  - Première phase : acquérir tous les verrous dont la tâche a besoin
  - Deuxième phase : relâcher les verrous à la fin des manipulations
- Règles
  - chaque objet / variable est protégée par des verrous (en écriture ou en lecture)
  - pour accéder à une variable, il faut demander implicitement un verrou
    - on ne peut pas donner un verrou en lecture ou en écriture sur une variable en cours d'ériture
    - on ne peut pas donner un verrou en écriture sur une variable en lecture
    - on peut accorder plusieurs verrous en lecture simultanée

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Problèmes liés à l'utilisation concurrentielle des variables

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Verrouillage deux phases : illustration

Exemple du code avec variables partagées

```
// poser des verrous ici

if ( A > 3)

     X = X + 1;

else

     Y = Y + 1;

// liberer les verrous
```

- Les verrous :
  - variable A : en lecture
  - variables X et Y : en écriture

Problèmes liés à l'utilisation concurrentielle des variables

Exemple : les verrous dans les BD SQL

- Les ressources critiques sont des données
- Les manipulations sont organisées en transactions
- Au début d'une transaction, on peut poser des verrous
- La lecture des valeurs présentes avant la transaction est toujours possible
- La fin de la transaction enlève les verrous

begin ;
lock table biere in exclusive mode ;
lock table bar in row share exclusive mode nowait ;
*select * from client ;*
delete from biere where quantite < 15 ;
...
commit ;

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Exclusion mutuelle

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Résumé

- L'exclusion mutuelle est une méthode qui permet à un processus ou thread d'utiliser une ressource critique et d'exclure les autres processus de cette activité

  La mise en oeuvre de l'exclusion mutuelle peut être variée

- La section critique dans un programme est la partie à partir de laquelle on accède à une ressource partagée et critique

  Pour éviter les problèmes concurrentiels, il faut empêcher que deux processus se retrouvent dans leurs sections critiques en même temps

Exclusion mutuelle

- **Exigeances** par rapport aux solutions
  - solution identique pour toutes les tâches
  - toute tâche reste un temps fini en section critique
  - une seule tâches en section critique
  - le blocage mutuel des tâches (*interblocage*) à l'entrée de la section critique doit être évité
  - une tâche bloquée hors de la section critique ne doit pas empêcher une autre tâche d'y entrer
  - une tâche qui se détruit en section critique ne doit pas bloquer l'usage de la section critique

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Réaliser l'exclusion mutuelle - première idée

Pour réaliser l'exclusion mutuelle, on peut utiliser des verrous

- Un verrou est un objet (une variable) qui possède une clé
- Son utilisation :
  - pour faire une activité, il faut demander et avoir la clé du verrou
  - si la clé n'est pas disponible, le demandeur doit (ou il peut) attendre que la clé se libère

Remarque : la généralisation pour des verrous à n clés est possible

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Moniteurs en Java - une solution simple

Java offre la possibilité de synchronisation des objets et des méthodes à l'aide du mot clé synchronized.

- Une section critique peut être verrouillée :
  - en utilisant un objet "verrou" ou moniteur (un seul thread peut avoir le verrou...)

```
synchronized(un_verrou) {
  ... //section critique
   }
```

  - en réalisant par une méthode (le verrou est l'objet courrant)

```
class une_classe{
 ...
public synchronized void methode() {
   ... //section critique
   }
}
```

- Un seul thread peut avoir le verrou d'une section critique englobée par cette synchronisation
- le verrou appartient à l'objet $\rightarrow$ deux objets d'une même classe doivent utiliser une méthode static

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Méthode synchronized, première exemple

```
public class ex12_partage extends Thread {
    private static String chaineCommune = "";
    private static int cpt = 0;
    private String nom;
    ex12_partage ( String s )  { nom = s;   }
    public void run() {
        for (int i = 0; i<10; i++)   {
         chaineCommune = chaineCommune + nom;
         cpt++;
         try {
              Thread.sleep(100);  // milliseconds
             } catch(InterruptedException e) {}
         }
    }
    public static void main(String args[]) {
         Thread T1 = new ex12_partage( "T1 " );
         Thread T2 = new ex12_partage( "T2 " );
         T1.start();
         T2.start();
         try {
              Thread.sleep(1000);  // milliseconds
             } catch(InterruptedException e) {}
         System.out.println( chaineCommune );
         System.out.println( cpt );
    }
}
```

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Méthode synchronized, résultats de la première exemple

```
miklos@miklos-laptop $ java ex12_partage
T2 T1 T2 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1
19
miklos@miklos-laptop $
```

Remarques :
Le compteur est à 19, mais il n' y a que 17 noms dans la chaine...

Comment expliquer les résultats ?

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Méthode synchronized, deuxième exemple

```
public class ex13_partage extends Thread {
  . . .
    public void run() {
       for (int i = 0; i<10; i++) {
          maj(nom); // mises a jour
        try {
             Thread.sleep(100);  // milliseconds
            } catch(InterruptedException e) {}
        }
    }
   public synchronized void maj(String nn){
                                  //section critique
        chaineCommune = chaineCommune + nn;
        cpt++;
   }
    public static void main(String args[]) {
        Thread T1 = new ex13_partage( "T1 " );
        Thread T2 = new ex13_partage( "T2 " );
        T1.start();
        T2.start();
        try {
             sleep(1000);  // milliseconds
            } catch(InterruptedException e) {}
        System.out.println( chaineCommune );
        System.out.println( cpt );
    }
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

Méthode synchronized, résultats de la deuxième exemple

```
miklos@miklos-laptop $ java ex13_partage
T1 T2 T1 T2 T1 T2 T1 T1 T2 T2 T1 T1 T2 T2 T1 T2 T1 T1 T2
19
miklos@miklos-laptop $ java ex13_partage
T1 T1 T1 T1 T1 T2 T1 T2 T1 T1
11
miklos@miklos-laptop $ java ex13_partage
java ex13_partage
T1 T2 T1 T2 T1 T1 T1 T2 T2 T2
10
miklos@miklos-laptop $
```

Remarques :

Pas d'amélioration...

Comment expliquer les résultats ?

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Méthode synchronized, troisième exemple

```
public class ex14_partage extends Thread {
  . . .
    public void run() {
       for (int i = 0; i<10; i++) {
          maj(nom); // mises a jour
        try {
             Thread.sleep(100);  // milliseconds
            } catch(InterruptedException e) {}
        }
    }
   public static synchronized void maj(String nn){
                                    //section critique
        chaineCommune = chaineCommune + nn;
        cpt++;
   }
    public static void main(String args[]) {
        Thread T1 = new ex14_partage( "T1 " );
        Thread T2 = new ex14_partage( "T2 " );
        T1.start();
        T2.start();
        try {
             sleep(1000);  // milliseconds
            } catch(InterruptedException e) {}
        System.out.println( chaineCommune );
        System.out.println( cpt );
    }
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Méthode synchronized, résultats de la troisième exemple

```
miklos@miklos-laptop $ java ex14_partage
T1 T2 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1
20
miklos@miklos-laptop $ java ex14_partage
T1 T2 T2 T1 T2 T1 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2
20
miklos@miklos-laptop $ java ex14_partage
T1 T2 T1 T2 T1 T2 T1 T2 T2 T1 T1 T2 T2 T1 T2 T1 T2 T1 T2 T1
20
miklos@miklos-laptop $
```

Remarques :

Comment expliquer les résultats ?

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Méthode synchronized, commentaires

- Plusieurs méthodes peuvent être synchronisées sur le même objet
- La synchronisation est ré-entrante, le *thread qui possède le verrou* peut entrer dans les blocs synchronisés

```
public class ex15_partage extends Thread {
  . . .

   public synchronized void m1(){
                                    //une section critique
        ...;
   }
    public synchronized void m2(){
                                   //une autre
        ...;
   }
}
```

- L'objet, le thread qui possède le "verrou" `this` pour `m1()`, peut aussi exécuter `m2()`
- Attention, la synchronisation d'une méthode statique et d'une autre non statique utilise deux verrous...

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Méthode synchronized, commentaires

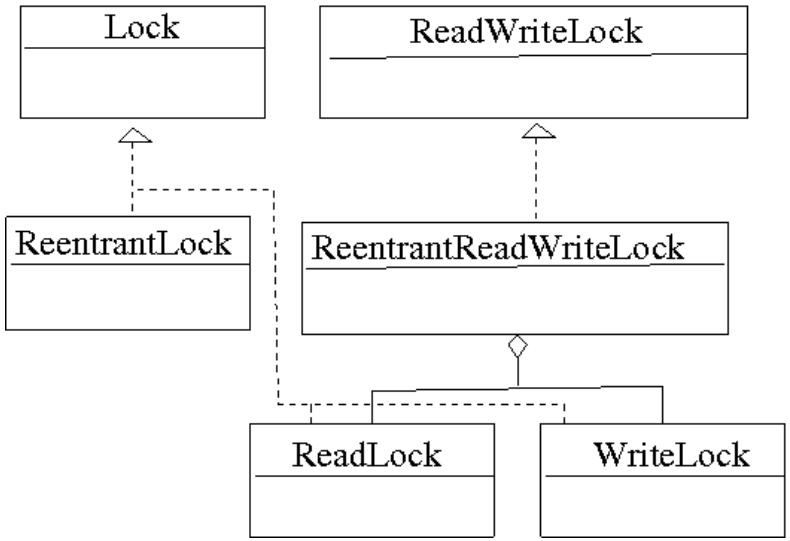
- Ce verrouillage est bloquant et ne fait pas de différence entre la lecture et l'écriture
  - en générale, la lecture d'une donnée est "thread-safe" (ne nécessite pas de synchronisation)
  - `synchronized` peut introduire des attentes si plusieurs threads veulent lire en même temps
- L'attente pour un verrou peut être illimité : pas de timeout
  - risque des deadlocks

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Verrous en Java - l'interface Lock

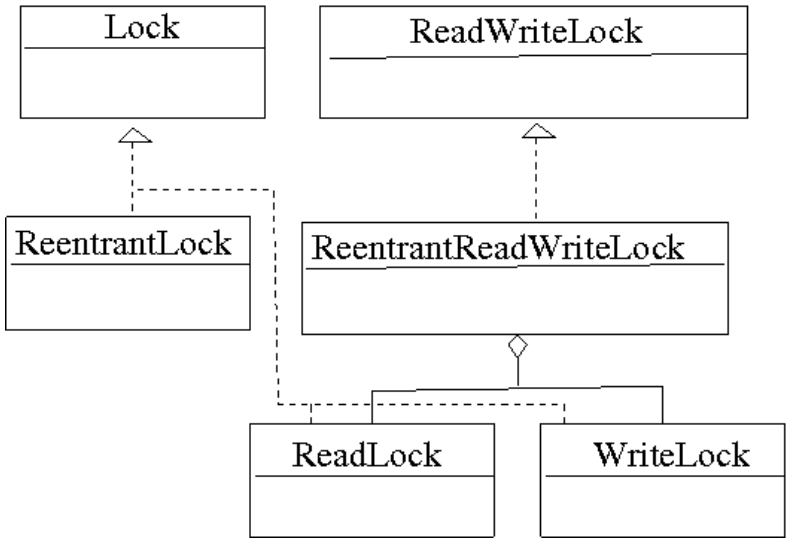


- Les méthodes "synchronized" utilisent les verrous implicites des objets, un seul objet peut avoir ce verrou
- Depuis Java 1.5 : l'interface Lock
  - Elle permet des structures plus fines et flexibles pour le contrôle
  - Deux méthodes : `lock()` et `unlock()`
  - Un objet implémentant Lock permet de régler l'accès à des ressources par plusieurs threads
    - souvent, il permet un accès exclusif (comme `WriteLock`)
    - mais quelques verrous permettent des accès concurrents comme le verrou `ReadLock` de la classe `ReadWriteLock`.

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Verrous en Java - l'interface Lock



Interface ReadWriteLock

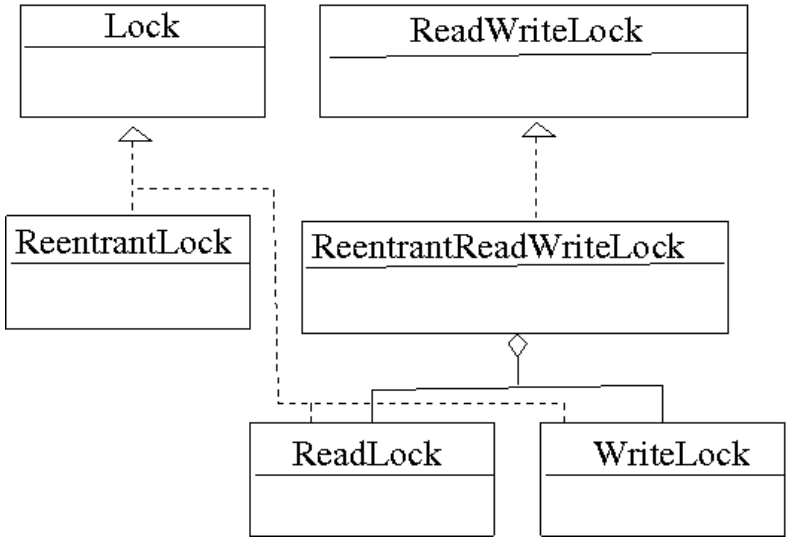
- Différente de Lock. Elle a deux méthodes :
  - readLock() : retourne un verrou en lecture
  - writeLock() : retourne un verrou en écriture
- Ces deux verrous sont des instances de Lock
- On peut avoir autant de verrous de lecture que l'on veut, tant qu'aucun verrou d'écriture n'a été demandé
- Le verrou d'écriture est exclusif : un seul thread peut l'avoir

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Verrous en Java - l'interface Lock



Deux classes instanciables

* Classe ReentrantLock
  - Implémente Lock
  - Dans le cas où un thread possède déjà le verrou et tente d'entrer dans un bloc synchronisé par le même objet qu'il possède

* Classe ReentrantReadWriteLock
  - Verrous réentrants
  - Il se compose de deux verrous

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Utilisation de la classe `ReentrantLock`

- On peut l'utiliser pour verrouiller (`lock()`) et déverrouiller (`unlock()`) explicitement
  - Un exemple

```
Lock l = new ReentrantLock();
l.lock();
 try {
  ...  //section critique
 } finally {
             l.unlock();
            }
```

- Un seul thread peut avoir le verrou d'une section critique englobée par cette synchronisation
- `unlock()` doit être dans un bloc finally
  - pour pouvoir toujours enlever le blockage

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Utilisation de l'interface Lock

- On peut associer des verrous à des objets et réaliser le verrouillage à deux phases
- Il faut créer des instances et invoquer les méthodes adéquates
  - Un exemple

```
creer des instances de type Lock
...
debut de section critique
  poser les verrous : lock() sur les instances
  manipulations
  liberer les verrous par unlock()
fin de section critique
```

- On veut poser les moindres verrous...

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Utilisation de l'interface ReadWriteLock

- Une variable partagée gardée avec `ReadWriteLock`
- Les threads concurrents :

```
public class RWLock implements Runnable {
...
private static int compteur =0;  // variable partagee
private static ReadWriteLock lck = new ReentrantReadWriteLock();
private static  Lock writeLock = lck.writeLock();// en ecriture
private static  Lock readLock = lck.readLock();  //en lecture
...
private Random rand = new Random();

public void run () {
  Random rand = new Random();
  for ( int i = 0; i < 10; i ++) {
    if (rand.nextInt(2) == 1) {
         incrementer(); }    // ecrire
    else{
        lire(); }     // lire la variable
      }
    }
```

Utilisation de l'interface ReadWriteLock

- Les méthodes concurrentes :

```
private void lire(){
      readLock.lock();          // verrou en lecture
      try{
         for ( int i = 0; i < 10; i ++) {
             System.out.println( RWLock.compteur );
             Thread.sleep(200);  // milliseconds
           }
          }catch(InterruptedException e) {}
         finally{
         readLock.unlock(); }  //on libere le verrou
    }
private  void incrementer () {
      writeLock.lock();          // verrou en ecriture
      try{
         //On fait le traitement 10 fois
         for ( int i = 0; i < 10; i ++) {
            RWLock.compteur = RWLock.compteur + 1;
            System.out.println( RWLock.compteur );
            Thread.sleep(200);  // milliseconds
          }
        }catch(InterruptedException e) {}
         finally{
         writeLock.unlock(); }  //on libere le verrou
    }
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

Exemple de la flexibilité de l'interface Lock

- On peut faire une tentative pour acquérir un verrou sans être bloqué s'il n'est pas disponible
  - Un exemple

```
Lock l = new ReentrantLock();
if (l.tryLock()) {
// ou encore l.tryLock(long time, TimeUnit unit)
     try {
       . . .    //section critique
      } finally {
                  l.unlock();
                 }
   }
   else {
      . . .   // actions alternatives
  }
```

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Synchronisation coopérative liée aux objets

Les méthodes suivantes permettent de coopérer pour synchroniser des threads.
Elles sont définies dans la classe `Object` (elles manipule le verrou associé à un objet), mais ne doivent s'utiliser que dans des méthodes `synchronized`

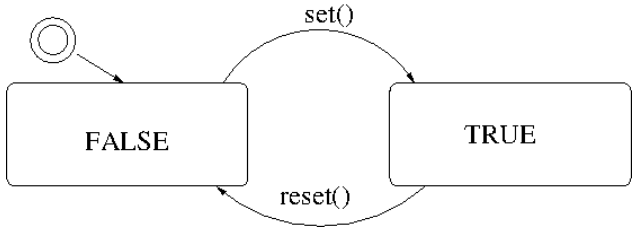
- `wait()` lève le verrou sur l'objet et bloque le thread appelant jusqu'à ce qu'une méthode `notify()` ou `notifyAll()` ne le réveille. Une interruption ou un timeout peut aussi conduire au déblocage du thread.
- `notify()` réveille un thread bloqué par un `wait()`. Si aucun thread n'est bloqué, rien ne se passe. Le thread débloqué doit alors prendre le verrou dès qu'il le pourra.
- `notifyAll()` réveille tous les threads bloqués. C'est le thread qui prend en premier le verrou qui accède à la section critique.

Miklós MOLNÁR

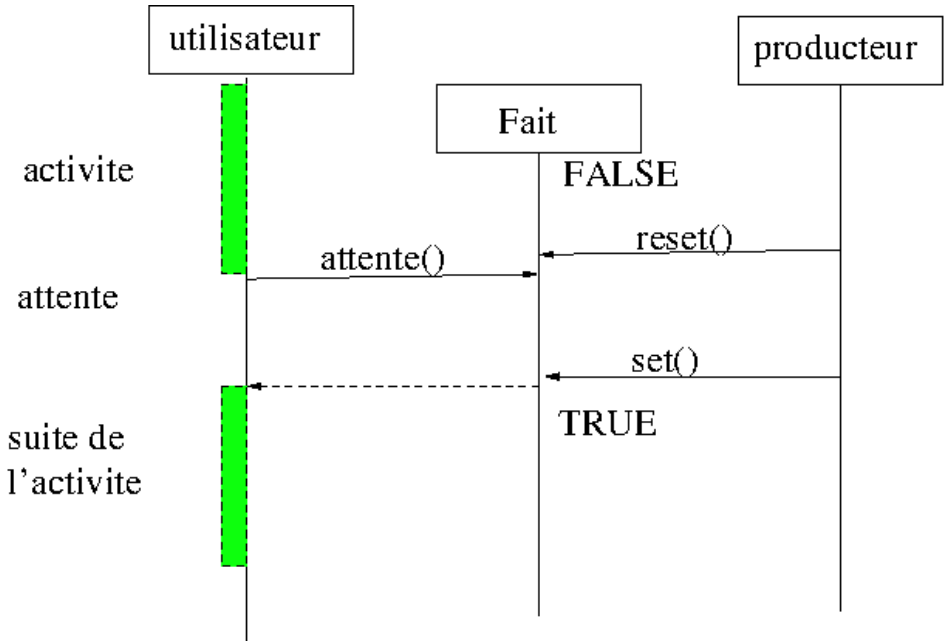
contacter
molnar@lirmm.fr

Exemple : une classe `Fait`

- L'automate d'un "fait"



- L'utilisation de la classe



Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Exemple : une classe `Fait`

```
public class Fait {
        private Boolean Etat; // etat du fait
        public Fait() {
                Etat = Boolean.FALSE;
        }
        public synchronized void set() {
                Etat = Boolean.TRUE;
        // debloque les threads qui attendent cet evenement
                notifyAll();
        }
        public synchronized void reset() {
                Etat = Boolean.FALSE;
        }
        public synchronized void attente() {
                if(Etat==Boolean.FALSE) {
                  try { wait(); // bloque jusqu'a un notify()
                  }
                  catch(InterruptedException e) {};
                }
        }
}
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

Réaliser l'exclusion mutuelle - deuxième idée

Pour réaliser l'exclusion mutuelle, on peut utiliser des sémaphores

- Un sémaphore est un objet (une variable) accessible uniquement à l'aide des opérations atomiques $P()$ et $V()$ (définition de Dijkstra)
- Les opérations :
  - $P()$
    - $S = S - 1$
    - si $S < 0$ alors le demandeur est bloqué dans une file spécifique au sémaphore
  - $V()$
    - $S = S + 1$
    - si $S \geq 0$ alors un des demandeurs bloqués dans la file est débloqué selon la politique appliquée

Remarques : la valeur initial de S (le nombre de jetons) détermine le nombre maximal des processus concurrentiels
La méthode $P()$ peut être bloquante mais pas $V()$

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Réaliser l'exclusion mutuelle - deuxième idée

Supposons un sémaphore avec un seul jeton

- Lorsqu'un thread veut entrer dans sa section critique, il appelle $P()$
  - S'il peut passer, il peut exécuter sa section critique
  - S'il ne peut pas, il attend qu'un jeton se libère
- Pour terminer, il appelle $V()$.

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Réaliser l'exclusion mutuelle - deuxième idée

- Sémaphore avec un seul jeton : *une implémentation possible*
  - à l'aide d'un tube
    - *P*() essaye de lire un caractère du tube, s'il n y en a pas, le processus est bloqué
    - *V*() dépose un jeton dans le tube (condition : pas de jeton dans le tube)

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Implémentations des sémaphores en Java

- Sémaphore à plusieurs jetons : une implémentation possible
  en utilisant le verrou implicite de l'objet (synchronized)

```
public class Semaphore {
        private int nb_jetons;
        private int nb_max;
        public Semaphore(int valeur, int limite) {
          nb_jetons = valeur;
          nb_max = limite; }
        public int getNbJetons() {
          return nb_jetons; }
        synchronized public void P() {
          while (nb_jetons < 1) {
            try { wait(); } // enleve le verrou !
            catch(InterruptedException e) {
                        System.out.println("Pb"); }
            }
          nb_jetons = nb_jetons-1;
        }
        synchronized public void V() {
          nb_jetons = nb_jetons + 1;
          notifyAll();
        }
}
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

Sémaphores avec valeur

- Ce sont des sémaphores à plusieurs jetons (qui permettent d'allouer plusieurs jetons à la fois)

```
public class Semaphore {
        private int nb_jetons;
        private int nb_max;
...
        synchronized public void P(int val) {
          while (nb_jetons < val)        { wait (); }
          nb_jetons = nb_jetons - val;
        }
//  wait() -> mise en file d'attente

        synchronized public void V(int val) {
          nb_jetons = nb_jetons + val;
          if (nb_jetons > nb_max)
                  nb_jetons = nb_max;
          notify();
// r\'eveiller des t\^aches  en attente
        }
}
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

La classe Semaphore depuis Java 1.5

- Une classe disponible

```
// import java.util.concurrent.Semaphore pour l'utiliser
class Semaphore {
...
   public void acquire()
    // Acquires a permit from this semaphore, blocking until
    // one is available, or the thread is interrupted.
   public void acquire(int permits)
    // Acquires the given number of permits
   public int availablePermits()
    // Returns the current number of permits available
   public Collection<Thread> getQueuedThreads()
    // Returns a collection containing threads waiting
    // to acquire
   public boolean isFair()
     // Returns true if this semaphore has fairness set
     // true.protected => FIFO order in the queue
   public void release()
     // Releases a permit, returning it to the semaphore
   public boolean tryAcquire()
     // Acquires a permit from this semaphore, only if
     // one is available at the time of invocation
...
}
```

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Utilisation de la classe Semaphore Java 1.5

- Exemple avec 1 jeton

```
import java.util.concurrent.Semaphore

Semaphore sem = new Semaphore(1);
...
    try {
        sem.acquire();
        //section critique
        sem.release();
    } catch(InterruptedException e) {
        e.printStackTrace();
    }
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

Sémaphores et verrous

- Un sémaphore à 1 jeton est très similaire à un verrou
- Cependant, ils sont légèrement différents
  - pour un sémaphore, si l'on effectue plusieurs fois l'opération V(), il garde et traite en mémoire les demandes (en incrémentant l'entier qu'il utilise)
  - pour un verrou, effectuer plusieurs déverrouillages est strictement équivalent à n'effectuer qu'un seul déverrouillage
  - (on peut ainsi voir plusieurs opérations V() successives sur un sémaphore comme des notify() "retardés" sur un verrou)

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Variables volatiles

- La valeur d'une variable `volatile` n'est pas placée dans une cache local d'une Thread
- Chaque lecture et écriture passe par la mémoire partagée entre les Threads.
- C'est toujours sa dernière valeur (volatile) qui est lue.
- Si la valeur peut être modifiée par un thread mais sans utiliser `synchronized` par les threads, la variable doit être volatile pour assurer que les threads puisse voir sa dernière valeur

Attention, `volatile` ne peut pas résoudre les écritures en concurrence

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Variables atomiques en Java

Variables atomiques

- Variables `volatile` qui permettent de les accéder (en consultation ou en modification) en exclusion mutuelle
- Exemples
  - AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference, AtomicLongArray, AtomicReferenceArray
- Gestion efficace grâce à l'utilisation des TAS (Test-and-Set)
  - Lire le contenu d'un emplacement pour un registre, et écrire une valeur à sa place
  - Cette paire d'opérations se fait de manière atomique
- La section critique est très courte
- Voir package `java.util.concurrent.atomic`

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Utilisation des variables ”atomiques”

- Exemple

```
public class Counter {
 private AtomicInteger count = new AtomicInteger(0);
 public void incrementCount() {
   count.incrementAndGet();
 }
 public int getCount() {
   return count.get();
 }
}
```

- Cette classe garantie qu’une seule tâche incrémente et/ou accède à la valeur

Miklós MOLNÁR

contacter
molnar@lirmm.fr