4. Outils pour la synchronisation

F. Boyer, Laboratoire Lig
Fabienne.Boyer@imag.fr

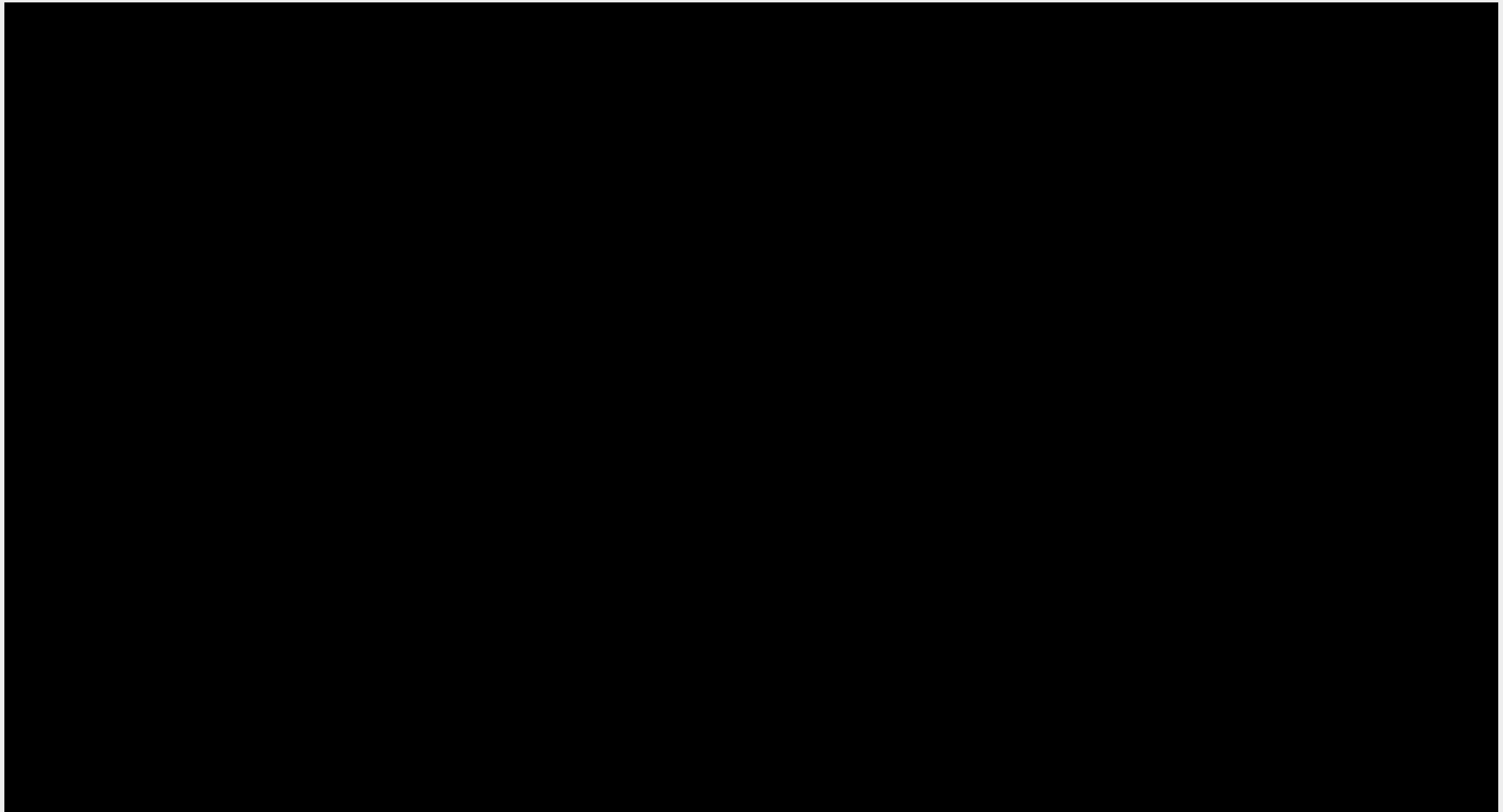
- **Le problème**
  - Insuffisance des solutions de base (verrous)
- **Les solutions de plus haut niveau**
  - Les Sémaphores
  - Les Moniteurs
  - Les Sections Critiques Conditionelles

Insuffisance des solutions de base pour la synchronisation

- **Les verrous représentent une solution simple pour gérer des sections critiques**

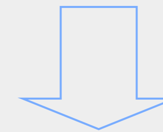
- **Ils ne permettent pas à eux-seuls de gérer des attentes passives *conditionnelles***

Exemple du parking



Solutions de plus haut niveau

- **Les principes : exploiter la sémantique de l'application pour**
  - Endormir un processus lorsqu'il ne peut pas continuer à s'exécuter
  - Le réveiller lorsqu'il peut continuer à s'exécuter



Attente passive conditionnelle

- **Les solutions :**
  - Les sémaphores
  - Les moniteurs
  - Les sections critiques conditionnelles

Sémaphores (Dijkstra, 1965)

Sémaphore *S* :
compteur S.c → Modélise un nombre de ressources ou une condition de passage
file d'attente S.f → Processus en attente

Fournit 2 opérations atomiques :
P → Prendre une ressource
V → Libérer une ressource

Sémaphores (Dijkstra, 1965)

Solution avec compteur initial positif ou nul

wait() ou P ():

```
    S.c--
    if (S.c < 0) do {
        // no more free resources
        put(current, S.f);
        suspend();  // suspension
}
```

signal() ou V ():

```
    S.c++;
    if (S.c <= 0) do {
      // at least 1 waiting process
        p = get(S.f);
        wakeup(p);
}
```

- **suspend()** suspend le processus sourant et libère la SC
- **wakeup(p)** remet p dans la file des prêts

sections critiques

Sémaphores (Dijkstra, 1965)

Exemple de mise en œuvre avec mask/unmask
(hyp: compteur initialement positif ou nul)

<u>wait() ou P ():</u>

```
    mask();
    S.c--
    if (S.c < 0) do {
      // no more free resources
          suspend(current, S.f);
    unmask();
}
```

<u>signal() ou V ():</u>

```
    mask();
    S.c++;
    if (S.c <= 0) do {
     // at least 1 waiting process
       wakeup(get_first(S.f));
    unmask();
}
```

Sémaphores

- Compteur S.c == S.c initial + NV - NP
  - NV est le nombre d'opérations V exécutées sur le sémaphore
  - NP est le nombre d'opérations P exécutées sur le même sémaphore
- Compteur S.c < 0 : correspond au nombre de processus bloqués
- Compteur S.c > 0 : correspond au nombre de ressources disponibles
- Compteur S.c == 0 : aucune ressource disponible et aucun processus bloqué

Application bancaire avec un sémaphore

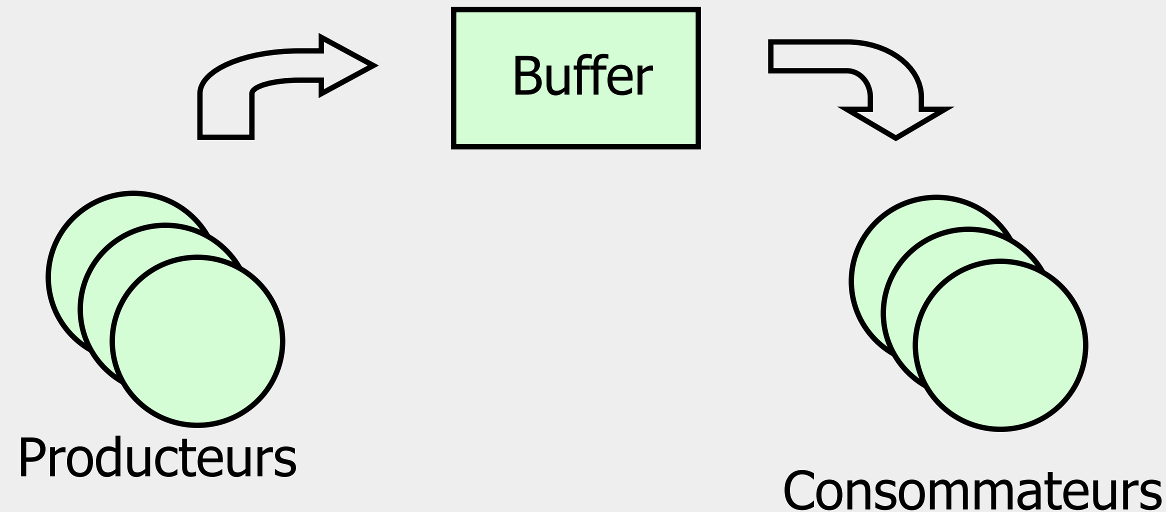
```
//shared variables
semaphore mutex = new Semaphore(1);

class Bank {
  …
  void credit(account, amount) :
    mutex.P();
    account= account+ amount;
    mutex.V();
  }
```

→ PB : séquentialisation des opérations qqsoit le compte

→ Utiliser un sémaphore par compte ?

Problème du producteur/consommateur



→ **Conditions de dépôt et de retrait**

- **Dépôt : buffer non plein**
- **Retrait : buffer non vide**

→ **Protéger les données partagées**

Producteur / Consommateur

- **Conditions de dépôt/retrait**
  - **Soit on utilise des variables pour évaluer les conditions de dépôt et de retrait, et on protège ces variables par un sémaphore**
    - int nbEmpty, nbFull;
  - **Soit on utilise des sémaphores pour représenter les conditions de dépôt et de retrait**
    - Semaphore notEmpty, notFull;

- **Protection des données partagées**
  - Sémaphore mutex

Producteur / Consommateur

Données partagées :

```
Msg buffer[] = new Msg[1];
// production condition
Semaphore notFull = new Semaphore(1);
// consommation condition
Semaphore notEmpty = new Semaphore(0);
// shared data protection
Semaphore mutex = new Semaphore(1);
```

Producteur / Consommateur

- **Processus Producteur :**

```
produce (Msg msg) {
        // if the buffer is full, wait until it becomes empty
        notFull.P();

        mutex.P();
        buffer[0] = msg;
        mutex.V();

       // wakeup some waiting process
        notEmpty.V();
}
```

Producteur / Consommateur

Processus Consommateur

```
Msg Consume {
    // if buffer is empty, wait until it becomes full
     notEmpty.P();

     mutex.P();
     Msg msg = buffer[0];
     mutex.V();

      // wakeup some waiting process
     notFull.V();
    return msg;
}
```

Producteur / Consommateur

Processus Consommateur

```
Msg Consume {
      // if buffer is empty, wait until it becomes full
       notEmpty.P();

       mutex.P();
       Msg msg = buffer[0];
       mutex.V();

        // wakeup some waiting process
       notFull.V();
      return msg;
}
```

Mutex nécessaire ?

Producteur / Consommateur

- **Gestion d'un buffer de N cases (N>1)**

Données partagées

```
int  bufferSz;
Msg buffer[];
Semaphore notFull;
Semaphore notEmpty;
Semaphore mutex;
int in = 0, out = 0;
```

Initialisation

```
public ProdCons(int bufferSz) {
    this.bufferSz = bufferSz;
    buffer = new Msg[bufferSz];
    notFull = new Semaphore(bufferSz);
    notEmpty = new Semaphore(0);
    mutex = new Semaphore(1);
}
```

Producteur / Consommateur

- **Processus Producteur, buffer à N cases (N>1) :**

```
Produce(Msg msg) {
        // if the buffer is full, wait until it becomes empty
        notFull.P();
        mutex.P();
        buffer[in] = msg;
        in = in + 1 % bufferSz;
        mutex.V();
        // wakeup some waiting process
        notEmpty.V();
}
```

Producteur / Consommateur

- **Processus Consommateur,buffer à N cases (N>1)**

```
Msg Consume() {
        // if buffer is empty, wait until it contains one message
        notEmpty.P();
        mutex.P();
        Msg msg = buffer[out];
        out = out + 1 % bufferSz;
        mutex.V();
        // wakeup some waiting process
        notFull.V();
}
```

Producteur / Consommateur

- **Pas de parallélisme entre productions et consommations**
  - **manque d'efficacité**
  - → **On essaie d'enlever le mutex**
  - → **Les opérations suivantes doivent rester exclusives**
    - in = in + 1 % bufferSz
    - out = out + 1 % bufferSz
  - → **Utiliser deux sémaphores dédiés**
    - Sémaphore mutexIn = new Semaphore(1);
    - Sémaphore mutexOut = new Semaphore(1);

Producteur / Consommateur

- **Processus Producteur, buffer à N cases (N>1)**

```
Produce(Msg msg) {
      // if buffer is full, wait for one empty entry
       notFull.P();
       mutexIn.P();
       buffer[in] = msg;
       in = in + 1 % bufferSz;
        mutexIn.V();
      // wakeup some waiting process
       notEmpty.V();
}
```

Producteur / Consommateur

- **Processus Consommateur, buffer à N cases (N>1)**

```
Msg Consume() {
        notEmpty.P();
        // if buffer is empty, wait for one item
        mutexOut.P();
        Msg msg = buffer[out];
        out = out + 1 % bufferSz;
        mutexOut.V();
        notFull.V();
        return msg;
}
```

Interblocages

- **Les sémaphores ne garantissent pas une solution correcte :**

```
P(mutex);
if ...
    P(S);
   ...
else
    ...
    V(S);
V(mutex);
```

→ **interblocage possible**

REGLE: jamais de blocage dans une SC sans libérer la SC

- **La gestion des interblocages est étudiée au chapître suivant**

Le moniteur

- **Module comprenant**
  - Des données
  - Des procédures d’accès (P1,..,Pn)
  - Une procédure d’initialisation
  - Des conditions

Les procédures sont exécutées en exclusion mutuelle

Une condition est une structure fournissant deux opérations :

- wait() bloque le processus courant
- signal() réveille un processus bloqué s’il y en a un. Le signal est fugace.

En général, les conditions sont gérées de manière FIFO

Schéma de moniteur

```
monitor <monitor-name> {
    <shared variables  + conditions declarations>

    procedure P1 (...) {
          . . .
    }
    procedure P2 (...) {
          . . .
    }
    procedure Pn (...) {
          . . .
    }
    { initialization code }
    }
```

Fonctionnement d'un moniteur

- Au maximum un seul processus actif dans le moniteur

- Lors d'un signal
  - Soit le signalant garde le moniteur (priorité signalant)
  - Soit le signalé prend le moniteur (priorité signalé)

- Libération du moniteur
  - Lorsque la procédure en cours est terminée
  - Lors d'un wait

- Lors de la libération du moniteur
  - Le moniteur peut être alloué en priorité à un processus déjà dans le moniteur (un signalé ou un signalant selon la priorité appliquée lors du signal)
  - Ou bien il n'y a pas de règle (tous les processus sont en concurrence) → vol de cycle possible

Moniteur Producteur/Consommateur

Hyp: moniteur à priorité signalant, FIFO, sans vol de cycle

```
Monitor ProdCons
  int bufferSz = 0;
  int nbMsg = 0;
  Msg buffer[];
  Condition notEmpty, notFull;
```

```
procedure init (int bufferSz)
        this.bufferSz = bufferSz;
        buffer = new Msg[buffersz];
```

```
procedure produce(Msg msg)
        if (nbMsg==BUFFER_SZ)
                notFull.wait();
        buffer[in] = msg;
        in = in + 1 % BUFFER_SZ;
        nbMsg++;
        notEmpty.signal();
```

```
procedure consume() : Msg
        if (nbMsg==0)
                notEmpty.wait();
        Msg msg = buffer[out];
        out = out + 1 % BUFFER_SZ;
        nbMsg--
        notFull.signal();
```

Moniteurs Java – rapide aperçu

- **Eléments synchronisés :**
  - **Instances**
  - **Classes**

- **Principes d'un moniteur**
  - **Méthodes synchronisées = exécutées en exclusion mutuelle**
  - **Opérations wait et notify/ notifyAll pour gérer des conditions**

```
class Example {

 int cpt;  // shared data

 public void synchronized get() {
       while (cpt <= 0) wait();
       cpt--;
}
public void synchronized put() {
       cpt++;
       notify();
}
}
```

Principes d'implantation des moniteurs Java

- **Tout objet synchronisé possède un verrou**
  - **Transparent / programmeur**
  - **Gestion FIFO / non FIFO dépend du JDK**

Méthodes synchronisées :

Méthode d'instance	Méthode de classe (static)
current.lock() <méthode> current.unlock()	current.lock() <méthode> current.unlock()
current = instance	*current = class*

Principes d'implantation des moniteurs Java

Tout objet synchronisé possède un verrou et une file de processus bloqués

```
wait()
    put(current_thread, blocked)
    current.unlock()
    suspend(current)
    current.lock()
```

```
notify()
  if !empty(blocked) {
      thread = get(blocked)
      wakeup(thread)
  }
```

Usage des moniteurs Java

- **Une seule condition (implicite)**
- **Vol de cycle possible**

```
class Parking
  int nfree = 0;

public Parking(int nplaces)
        this.nfree = nplaces;

public synchronized void enter(){
              while (nfree == 0)
                          this.wait();
              nfree--;
}

public synchronized void leave(){
       nfree++;
       this.notify();
}
```

Usage des moniteurs Java

```
class Parking          public Parking(int nplaces) {this.nfree = nplaces;}
  int nfree = 0;

public synchronized void enter(){
        while (nfree==0 ÍÍ nprio>0)
                this.wait();
        nfree--;
}

public synchronized void enterPrio(){
     nprio++;
     while (nfree==0)
           this.wait();
      nfree--; nprio--;
}

public synchronized void leave(){
      nfree++;
      this.notify();
}
```

Moniteur Producteur/Consommateur

Hyp: moniteur à priorité signalant, FIFO, avec vol de cycle

```
class ProdCons {
  int bufferSz = 0;
  int nbMsg = 0;
  Msg buffer[];

ProdCons (int bufferSz){
      this.bufferSz = bufferSz;
      buffer = new Msg[buffersz];
  }

synchronized void produce(Msg msg){
        while (nbMsg==BUFFER_SZ)
              wait();
        buffer[in] = msg;
        in = in + 1 % BUFFER_SZ;
        nbMsg++;
        notifyAll();
   }

synchronized Msg consume(){
      while (nbMsg==0)
            wait();
      Msg msg = buffer[out];
      out = out + 1 % BUFFER_SZ;
      nbMsg--
      notifyAll();
}
```

Usage des moniteurs Java

- **Gestion des interruptions**
  - **Tout thread bloqué peut être interrompu (méthode interrupt() de la classe Thread)**
  - **L'appel *wait()* peut lever l'exception InterruptedException**
  - **Principe: boucler sur la condition applicative**

```
public void synchronized M() {
        …
        while (<condition>) {
                try {
                   wait();
                } catch (InterruptedException e){ }
        }
        …
}
```

Moniteurs Java

```
class Parking
  int nfree = 0;

public Parking(int nplaces)
        this.nfree = nplaces;

public synchronized void enter(){
            while (nfree == 0)
                  try {
                    this.wait();
                  catch (InterruptedException e) {..}
            nfree--;
}
```

```
public synchronized void leave(){
        nfree++;
        this.notify();
}
```

Les sections critiques conditionnelles

- Outil de synchronisation POSIX (verrou + condition)
- Gestion explicite de l'exclusion mutuelle

  - lock(verrou)
  - wait(verrou, condition)
  - signal(verrou, condition)
  - unlock(verrou)

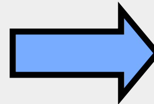
→ wait(V, C) libère le verrou V, bloque le processus courant sur C, puis reprend le verrou V à son réveil

→ signal(V, C) réveille un processus bloqué sur C, s'il y en a un (fugace)

→ En général, verrous et conditions FIFO

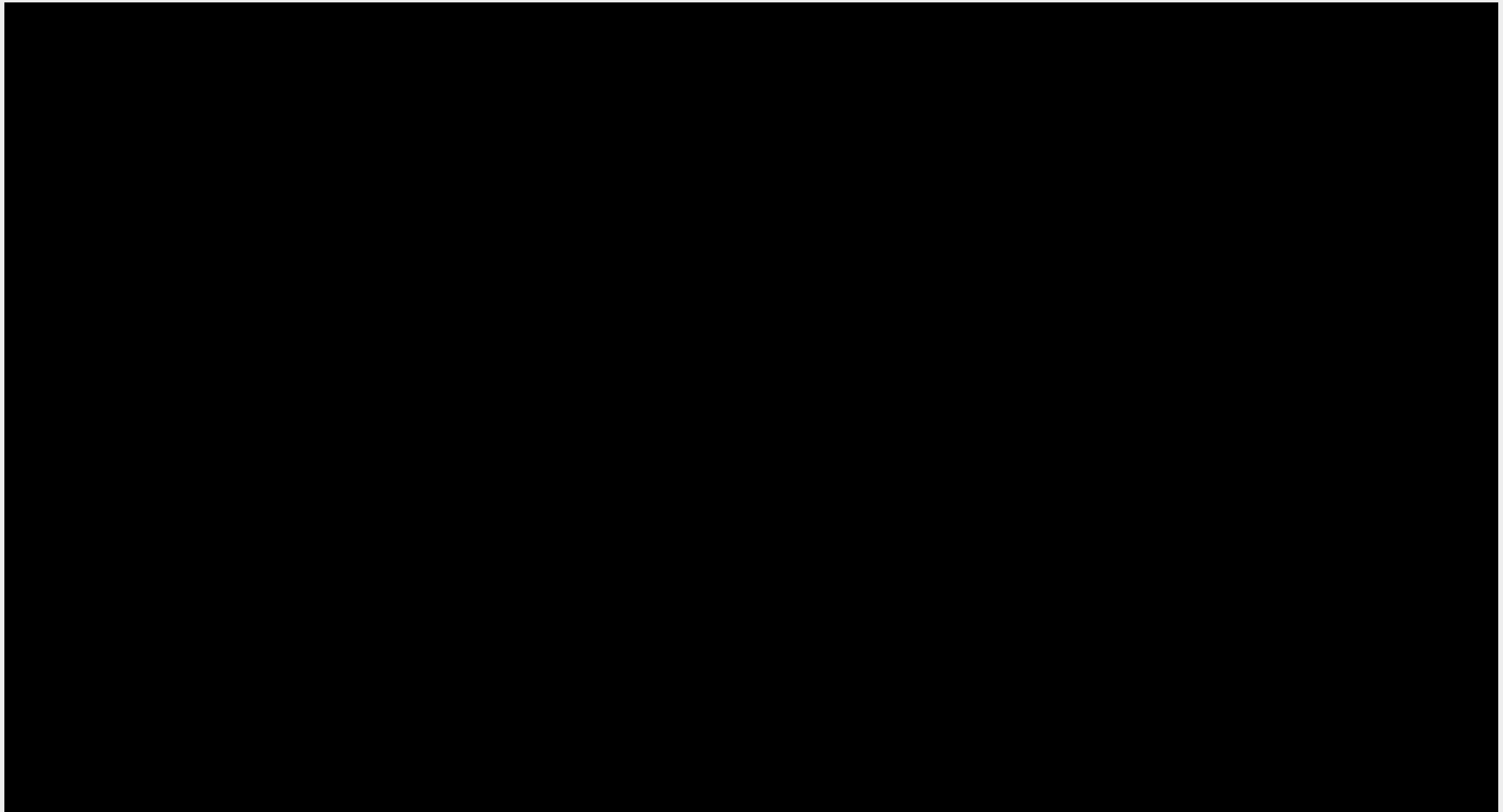
SCC « Moniteur » (Implantation procédurale d'un moniteur avec une SCC)

```
monitor M {
    condition c;
    procedure P (…) {
        …
        wait(c)
        . . .
        signal(c)
    }
}
```



```
SCC M {
    condition c;
    lock l;
    void P (…) {
        lock(l);
        . . .
        wait(c, l);
        . . .
        signal(c,l)
        . . .
        unlock(l);
    }
}
```

SCC Parking



Implantation des SCC
priorité signalant – avec vol de cycles

```
class Scc {
Lock mutex;
Lock condition; // basic implementation (only manages 1 condition)
int nwaiting;   // number of processes waiting on the condition

public Scc {
  mutex       = new Lock(); // manages the SCC mutual exclusion
  condition  = new Lock(); // manages a FIFO waiting queue of threads
  condition.lock();       // ensures that next calls to condition.lock() will block
  nwaiting = 0;
}

public void lock () {
        mutex.lock();
}
public void unLock () {
        mutex.unLock();
}
```

Implantation des SCC (2)
priorité signalant – avec vol de cycles

```
public void wait() {
        nwaiting++;
        mutex.unLock();              // free the SCC
        condition.lock();            // block the current process
        mutex.lock();                // re-enter in the SCC
}
public void signal() {
         if (nwaiting > 0) {
              nwaiting--;
             condition.unLock();     // wake up a waiting process
        }
}
public void signalAll() {
         while (nwaiting > 0) {
              nwaiting--;
              condition.unLock();
        }
 }
```

Implantation des SCC à base de verrous priorité signalant – sans vol de cycles

```
class Scc {

Lock mutex;
Lock condition;
int nwaiting;   // number of processes waiting for the SCC
Lock wakeup;  // to manage the coming back of the wakeUp processes in the SCC
int nwakeup;   // number of wakeUp processes

public Scc {
        mutex       = new Lock();
        condition   = new Lock();
        wakeup      = new Lock();
        condition.lock();     // ensure that upcoming calls to condition.lock() will block
        wakeup.lock();      // ensure that upcoming calls to wakeUp.lock() will block
        nwaiting = 0;
        nwakeUp = 0;
}
```

Implantation des SCC (2)
priorité signalant – sans vol de cycles

```
public void lock () {
        mutex.Lock();
}

public void unLock () {
  if (nwakeup > 0) {    // give the SCC to a wakeup process
            nwakeup --;
             wakeup.unLock();

  } else {
            mutex.unLock();  // free the SCC
       }
}
```

Implantation des SCC (3)
priorité signalant – sans vol de cycles

```
public void wait() {
        nwaiting++;
        this.unLock();
        condition.lock();  // block the current process in the condition waiting-set
        wakeup.lock();     // block the current process in the wakeup waiting-set
}
public void signal() {
         if (nwaiting > 0) {
              nwaiting--;
             condition.unLock();
             nwakeup++;
 }
public void signalAll() {
          while (nwaiting)> 0) {
               nwaiting--;
               condition.unLock();
               nwakeup++;
         }
}
```

Exclusion mutuelle

- Nous avons vu les outils

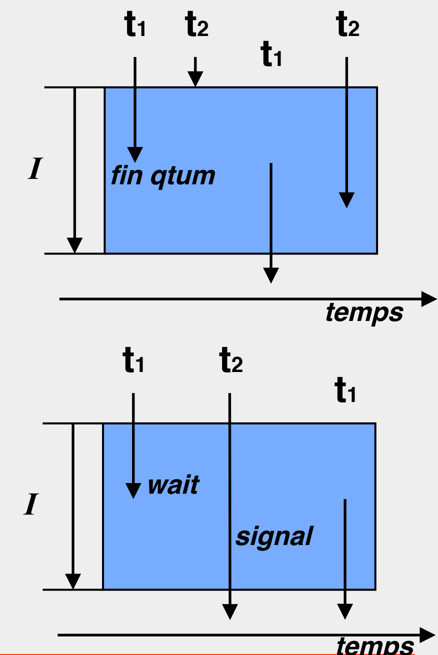
- Nous avons considéré des exemples élémentaires
  - Verrouillage d'un unique objet (parking)
  - Verrouillage d'un tableau (prod-cons)

- Considérons maintenant les choses sous un angle plus général..

Exclusion mutuelle et atomicité

Atomicité d'une suite d'instructions I → 1 seul thread/processus dans I

- **Atomicité par non-interruptibilité matérielle**
  Toute instruction assembleur
- **Atomicité par sauvegarde de contexte**
  L'instruction *int i=j;* en java est interruptible,
  mais a un comportement atomique
- **Atomicité par empêchement des commutations**
  Par ex, usage de mask() et unmask()
- **Atomicité via une section critique**
  L'usage de verrous force la sérialisation des exécutions de I
  On garantit qu'il y a au maximum un thread dans I
- **Atomicité via une section critique conditionnelle**
  On garantit qu'il y a au maximum un thread **non bloqué** dans I



Atomicité et cohérence mémoire

Atomicité ne signifie pas cohérence (visibilité)

- **Visibilité d'une variable :** la dernière valeur de la variable est visible par tout thread qui accède à la variable.

→ Pas toujours le cas lorsque les variables sont mises en cache localement à un thread ou à un CPU, ou sont gérées de manière optimisée par le compilateur

- **Visibilité provoquée sur l'entrée et la sortie d'un bloc synchronisé (ex: Java)**

  Les dernières valeurs des variables sont propagées à tous les threads

  Pas d'optimisation par le compilateur

- **Visibilité basée sur les variables *volatile***

  Variables (primitives) non optimisées et non mises dans un cache local

  Lectures/écritures atomiques

Java synchronization

- **From the JSR:**

After we exit a synchronized block, we release the monitor, which has the effect of flushing the cache to main memory, so that writes made by this thread can be visible to other threads.

Before we can enter a synchronized block, we acquire the monitor, which has the effect of invalidating the local processor cache so that variables will be reloaded from main memory. We will then be able to see all of the writes made visible by the previous release.

Usage de variables volatiles en Java (exemple)

```
class X () {
        static boolean ready = false;
        void waitReady() { while (!ready) ;}
        void setReady() { ready = true; }
}
```

```
Thread T1:
        …
        X.waitReady();
        …
```

```
Thread T2:
        …
        X.setReady();
```

➔ T1 peut rester bloqué indéfiniment sur *waitReady()*

➔ Pour éviter cela, définir la variable *ready* comme *volatile*

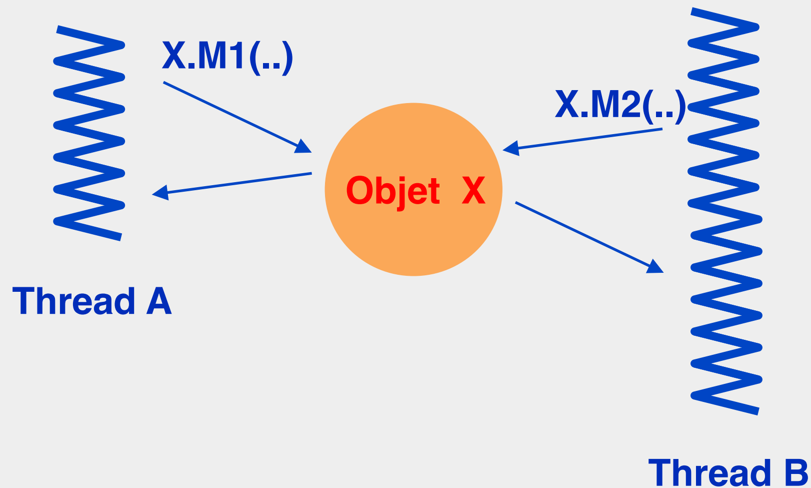
Thread-safety

- ***Qu'est-ce qu'une classe "thread-safe"***
  - A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.
  - Stateless objects are always thread safe.

  ref: Java Concurrency in Practice

- **Comment obtient-on une classe thread-safe?**
  - **Garantir atomicité et visibilité pour les données partagées**
  - Utiliser les directives synchronized et/ou volatile
  - Attention: assigner plusieurs variables volatile n'est pas globalement atomique
  - Attention: lire puis écrire une variable volatile n'est pas globalement atomique

Exclusion mutuelle et aliasing

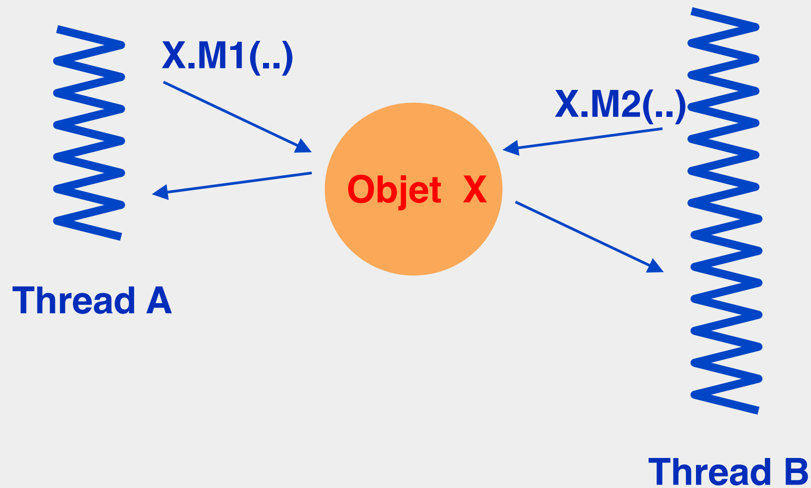


Questions

- **Faut-il synchroniser les 2 invocations?**
- **Si oui, comment les synchroniser?**

- A et B possèdent la référence de l'objet X
- L'objet X peut être une instance ou une classe

Exclusion mutuelle et aliasing

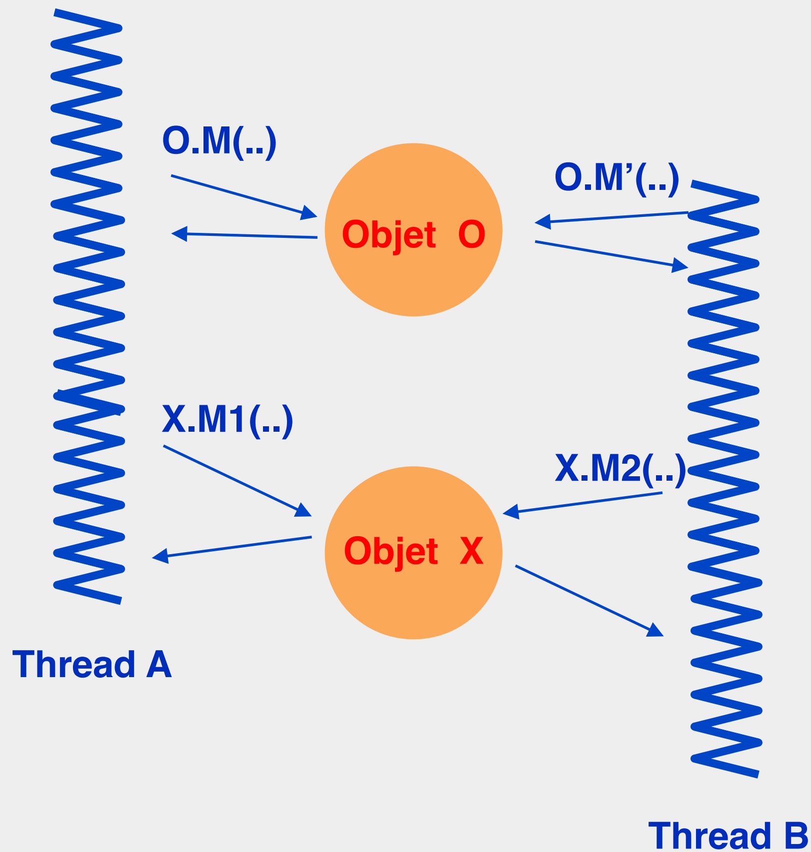


Solution 1

M1 et M2 sont définies comme des méthodes exclusives (e.g., synchronisées)

ou bien M1 et M2 contiennent des blocs d'instructions synchronisés

Exclusion mutuelle et aliasing



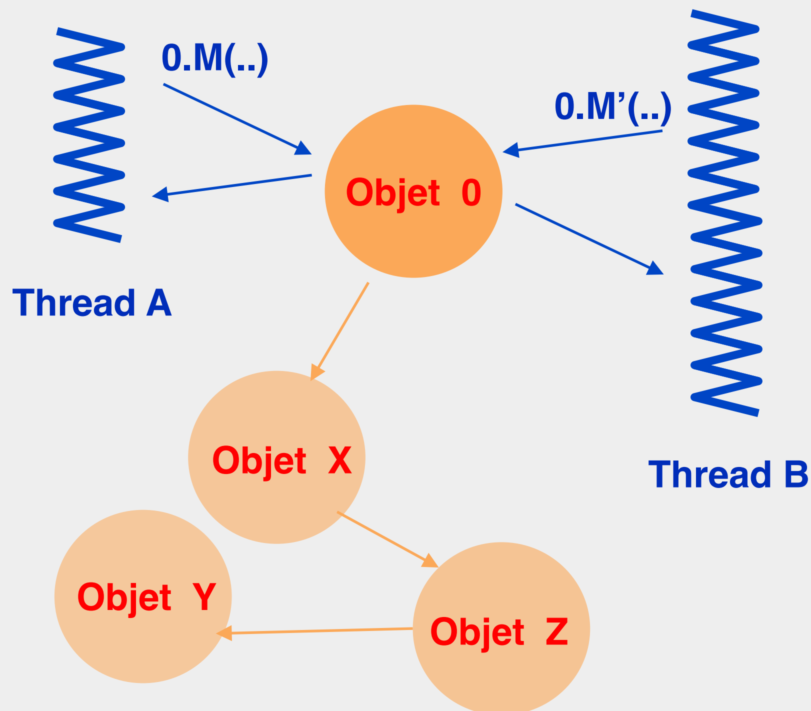
Solution 2

L'accès à X passe par un objet frontal qui est synchronisé

Attention

- Cela suppose que tous les threads respectent cette règle

Exclusion mutuelle et aliasing



Question

-Faut-il synchroniser O et les objets accessibles depuis O?

Réponse

-Cela dépend de l'aliasing sur X, Y, Z

-Dans tous les cas, attention à l'imbrication de blocs synchronisés

Synchronisation Java

- **Basée sur les moniteurs**
  - Objets synchronisés
  - Threads + Données mutables partagées + Verrous
- **Java 5: package java.util.concurrent**
  - Variables atomiques
  - Sémaphores
  - Verrous à timeout
  - Collections spécialisées
  - Gestion de tâches asynchrones (Executors)
- **Java 7: FJTasks**
  - Tâches de type fork/Join
  - Gestion de calculs intensifs de type *recursive divide and conquer*

Classes non « thread-safe »

- **Listes**
  - Interface List<E>
    - ArrayList<E>
    - LinkedList<E>
    - CopyOnWriteArrayList<E>
- **Ensembles**
  - Interface Set<E>
    - HashSet<E>
    - TreeSet<E>
- **Associations**
  - Interface Map<K,V>
    - HashMap<K,V>
    - TreeMap<K,V>)

Les méthodes ne sont pas synchronisées

On doit en tenir compte dans notre programmation

Usage de classes non thread-safe en Java

```
class X () {
  ArrayList list;
  void M() {
    for (int i=0; i < list.size();) {
           if (testSomeCondition(list.get(i)))
             list.remove(i);
           i++;
      }
   }
..
```

ArrayList Javadoc:

Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.
(A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.)

Usage de classes non thread-safe en Java

Solutions

→ Wrapper la liste dans une liste synchronisée via la méthode Collection.synchronizedList. Faire cela au moment de la création de la liste pour éviter tout accès non voulu à la liste non synchronisée.

```
List slist = Collections.synchronizedList(new ArrayList(...));
```

→ Pour une synchronisation plus globale, définir des blocs synchronisés sur l'objet *list*, or sur tout objet encapsulant l'objet *list*

```
class X () {
 ArrayList list;
 void M() {
  synchronized(list) {
   for (int i=0; i < list.size();) {
          if (testSomeCondition(list.get(i)))
             list.remove(i);
          i++;
   }
  }
..
```

Classes Thread-safe

- **Vector, Hashtable,..**
  - Toutes les méthodes sont synchronisées (attention, cela ne fournit pas une synchronisation globale)
  - Si une modification est effectuée en parallèle avec l'utilisation d'un d'un itérateur, une exception est levée
  - Attention, ces classes peuvent générer des « bottleneck » en cas de forte concurrence

- **SynchronizedMap (Hashmap wrapping)**
  - Toutes les méthodes sont synchronisées (~Hashtable)

- **ConcurrentHashMap**
  - Optimisée pour la concurrence (lectures sales, écritures partitionnées)

Collections et itérateurs

- **Un itérateur *fail-fast* lève une exception si la collection est modifiée concurrement**
  - Contexte mono-thread
    - Exception si la collection est modifiée par toute méthode non liée à l'itérateur
  - Contexte multi-threads
    - Exception si la collection est modifiée par tout thread autre que le thread courant

- **Un itérateur *fail-safe* itère sur une copie de la collection**
  - Peut être coûteux (notamment si pas de mécanisme de copy-on-write ou si trop de writes..)

Propriétés des Collections Java

Property	Hashmap	Hashtable	SynchronizedMap	ConcurrentHashmap
Thread-safe	no	yes	yes	yes
locking	none	lock whole map	lock whole map	lock bucket
iterator	fail-fast	fail-fast	fail-fast	fail-safe
When to use	1 writer max	global consistency	global consistency	performances

Package java.util.concurrent

- **Classe Semaphore**
  - acquire / tryAcquire
  - release
- **Interface Lock**
  - lock / tryLock(timeout)
  - unlock
  - newCondition
- **Interface Condition**
  - await: le thread courant est bloqué jusqu'à ce qu'il soit signalé ou interrompu
  - signal / signalAll: réveille un thread bloqué
- **Classes ReentrantLock, ReentrantReadWriteLock**

ReentrantLock: schéma d’usage

- Garantir qu’un verrou est systématiquement relâché

```
class C {
    private final ReentrantLock l = new ReentrantLock();
    ...
   public void m() {
      l.lock();
      try {
          ...
      } finally {
         l.unlock();
      }
}
...
```

Blocking Queues

- **Interface BlockingQueue**
  - Type Producteur / consommateur

- **Classes**
  - **ArrayBlockingQueue** (basée sur l'usage d'un tableau borné)
  - **DelayQueue** (élement peut être retiré au bout d'un délai)
  - **LinkedBlockingQueue** (élements ordonnés FIFO, borné)
  - **PriorityBlockingQueue** (élements ordonnés / priorité)
  - **ConcurrentLinkedQueue** **(**file non bornée)
  - ...

Package java.util.concurrent (cont.)

Variables atomiques

- AtomicBoolean
- AtomicInteger
- AtomicLong
- AtomicReference
- AtomicLongArray
- AtomicReferenceArray

Getters et setters avancés

- Implémentés sans verrous
- Basés sur l'usage de volatile

Exemple

```
AtomicInteger i = new AtomicInteger(0);
int current = i.incrementAndGet();
```

Wrap Up

- Synchronized
- Wait
- Imbrication des Synchronized
- InterruptedException
- Hooks/Crochets cycle de vie JVM

Clause synchronized

```
synchronized(x){        <-- x.lock()
  …
}                       <-- x.unlock()
```

```
synchronized(x){
  …
  x.wait();       <-- x.addInWaitingSet(currentThread);
                      x.unlock();
                      context-swiych();
                      x.lock();
  ...
}
```

```
synchronized(this){
  …
  wait();         <-- this.unlock();
                      this.addInWaitingSet&commut(currentThread);
                      this.lock();
  ...
}
```

Classes Thread-safe

class List TODO

x.lock()

Attention à l'imbrication des clauses synchronized

```
class A{
    B b;
      ..
    static synchronized void M1(){
            ..;  b.W(); ..
    }
    static synchronized void M2(){
             ..; b.N(); ..
    }
 }
```

```
class B{
    public int x = 10;
    synchronized void W(){
                ..; wait(); ..
    }
    synchronized void N(){
                ..; notify(); ..
    }
   ..
```

blocage infini du à l'imbrication des blocs synchronisés
un thread qui se bloque ici n'a pas relâché le verrou pris sur A

Wait et InterruptedException..

InterruptedException: peut être levée dans toute méthode bloquante, soit par l'application (Thread t; .. t.interrupt()), soit par la JVM.

→ Lorsque wait() est utilisé pour attendre une garde, il suffit de catcher l'exception en général.

```
synchronized(x){
  …
   while !<garde> {
      try { x.wait(); } catch (InterruptedException e) {}
   }
  ...
}
```

→ Si toutefois l'application veut pouvoir interrompre un thread bloqué, il faut utiliser un flag.

```
synchronized(x){
  …
   while !<garde> {
      try { x.wait(); } catch (InterruptedException e) { if (appliStopped) return; }
   }
  ...
}
```

variable globale

InterruptedException de manière générale

Thread t; .. t.interrupt(); ..

→ Si t est bloqué (wait(), join(), sleep(), ..), t est débloqué et reçoit InterruptedException
→ Sinon le status d'interruption est positionné pour t

public static boolean interrupted()
si le statut d'interruption a été positionné – réinitialise ce statut à faux et renvoie vrai
sinon renvoie faux

public boolean isInterrupted()
renvoie la valeur du statut d'interruption

```
Thread t=new Thread(new Runnable() {
  public void run() {
      for (..;!Thread.interrupted();..) {..}
  }
 });
```

Crochets d'arrêt / Hooks

→ Possibilité d'enregistrer des crochets d'arrêt (Runtime.getRuntime().addShutdownHook(t), Runtime.getRuntime().removeShutdownHook(t)), t étant un objet de classe Thread.

→ Au moment de l'arrêt, la JVM invoque start() sur l'objet t. Si plusieurs hooks sont enregistrés, ceux-ci sont exécutés dans un ordre indeterminé.

```
..
Thread hook=new Thread(new Runnable() {
      public void run() {
          try{
              System.out.println("Stopping execution, saving all data on files");
              saveData();
              System.out.println("Done")}

          } catch(Exception e){
              e.printStackTrace();
          }
      }
});
Runtime.getRuntime().addShutdownHook(hook);
..
```

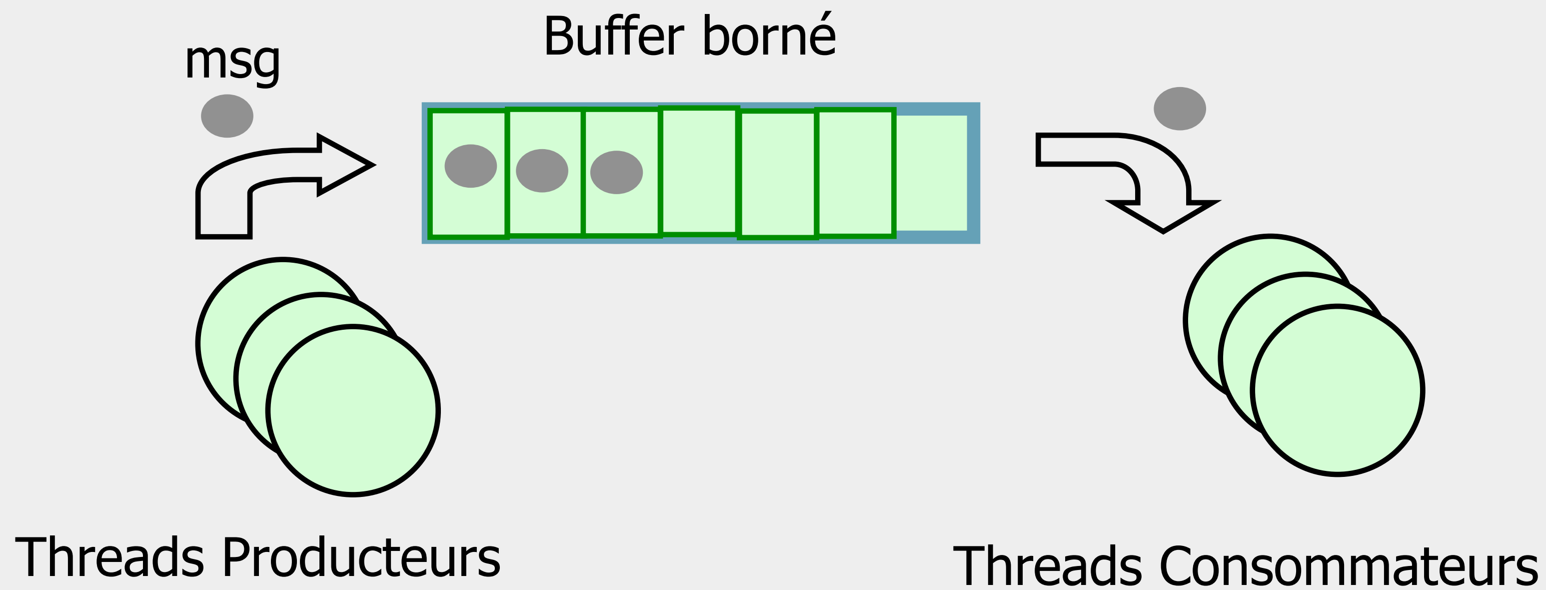
Codage des sémaphores en Java (<1.5)

```
Class Semaphore {  // ne fonctionne que pour des initialisations >= 0
 private int count;

 public semaphore(int count) {
   this.count = count;}

 public synchronized void P() throws Exception {
  if ((--count) < 0)
     try { wait=true;
        while (wait) {wait(); wait=false;}
     catch (InterruptedException e) {wait = true};
 }

 public synchronized void V() throws Exception {
  if ((++count) <= 0)
     notify();
 }
}
```


msg
Buffer borné
Threads Producteurs
Threads Consommateurs