COURS PARALLELISME

Mécanismes de synchronisation

Jean-Louis PAZAT

Plan

- Introduction
- Différents mécanismes de synchronisation
  - Sémaphores (rappels)
  - Régions critiques conditionnelles
  - Moniteurs
  - Compteurs de synchronisation
  - Evénements/ futurs
- Mécanismes de synchronisation dans les langages de programmation
  - Java
  - Threads Posix
  - OpenMP
- Conclusion

Introduction : Communication par partage de variables

- 2 classes de variables au sein d 'un programme parallèle:
  - les variables privées
    - un seul processus les connaît/ y accède
  - les variables partagées
    - accessibles par plusieurs/tous les processus

Introduction : Communication par partage de variables

- Accès aux variables partagées
  - Actions en général non atomiques
    - **accès** (lecture / écriture) à une variable tableau
    - **incrémentation** d'une variable scalaire
    - **Relations** à maintenir entre plusieurs variables
  - Exécution parallèle d'actions non atomiques
    - comportement non spécifié
    - état incohérent du système

Introduction : Communication par partage de variables

`x = x + 1 // x = x + 1`

```
load reg1, mem x
add  reg1, 1
store mem x, reg1
```

```
load reg2, mem x
add  reg2, 1
store mem x, reg2
```

```
load reg1, mem x
load reg2, mem x
add  reg1, 1
add  reg2, 1
store mem x, reg1
store mem x, reg2
```

x = x + 1

```
load reg1, mem x
add  reg1, 1
store mem x, reg1
load reg2, mem x
add  reg2, 1
store mem x, reg1
```

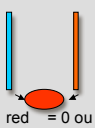
x = x + 2

Introduction : Communication par partage de variables

`mycolor.put(255,255,255)` `mycolor.put(0,0,0)`

red = 255
green=255
blue =255

red = 0
green=0
blue =0



red = 0 ou 255
green= 0 ou 255
blue = 0 ou 255

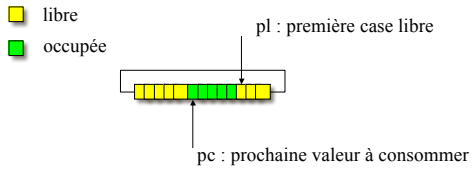
red = 255
green=0
blue =0

Introduction : Communication par partage de variables

- Maintenir la cohérence des variables
  - définir des actions atomiques
  - définir des invariants
    - sur les valeurs des variables
  - restreindre le parallélisme
    - pour rendre certaines actions atomiques
    - pour respecter les invariants

synchronisation

Introduction : Communication par partage de variables




```
Pre {Produire}: non plein(tampon)
Post{Produire}: non vide (tampon)

I1 = (libre (T[pl]) || plein(tampon))
      &&
     (occupée (T[pc]) ou vide(tampon))
```

Introduction : Communication par partage de variables

- Exemple de synchronisations possibles :
  - plein
    - seule Pre{Consommer} est vraie
  - vide
    - seule Pre{Produire} est vraie
  - ni plein ni vide
    - Pre{Produire} et Pre{consommer} sont vraies
    - parallélisme possible (en théorie)
    - mais restriction d'accès exclusif à la structure

Introduction : Communication par partage de variables

- Paradigmes de synchronisation
  - exclusion mutuelle
  - producteur/consommateur
  - lecteur/rédacteur
- Mécanismes de synchronisation
  - outils pour réaliser la mise en œuvre
    - sémaphores, moniteurs, ...

Introduction: paradigmes de synchronisation

- Exclusion mutuelle
  - définition
    - au plus un processus dans la section critique
    - tout processus qui demande l'accès à la section critique l'obtient au bout d'un temps fini
  - remarque
    - l'ordre d'accès à la section critique ne respecte pas nécessairement l'ordre des demandes

Introduction: paradigmes de synchronisation

- Producteur/consommateur
  - définition
    - une valeur ne peut être consommée qu'après avoir été produite
    - une valeur ne peut être consommée qu'une fois
  - remarque
    - la production d'une valeur ne nécessite à priori pas de synchronisation
    - c'est la structure de données qui l'impose
      - tampon infini, borné à N cases, sans tampon ...

Introduction: paradigmes de synchronisation

- Lecteur Rédacteur
  - définition
    - plusieurs processus peuvent accéder à une structure de donnée en parallèle en lecture
    - 1 et 1 seul processus peut accéder à la structure de données pour réaliser une écriture
    - lecture et écriture sont mutuellement exclusives
    - les écritures sont mutuellement exclusives
  - remarque
    - plusieurs lectures parallèles sont en général souhaitables

Introduction

- Remarques
  - Note sur la syntaxe [IMPORTANT]
    - la plupart des mécanismes présentés ont été introduits sous forme de modules ou intégrés à des langages procéduraux. Nous avons parfois utilisé une syntaxe “à la Java” uniquement pour faciliter la lecture.
  - Héritage et synchronisation
    - Les problèmes liés à la synchronisation et à l’héritage ne sont pas traités ici

Sémaphores [E.W. Dijkstra 68]: entier + file d’attente

```
class Semaphore {//depuis Java 1.5
  private int s;
  public Semaphore (int permits) // créé le sémaphore
   {s = permits;}
  public Semaphore (int s, boolean fair)
   // initialise le sémaphore
   // si fair = true les threads sont gérés dans une
  // FIFO (<=> Sémaphores habituels)
  public  void aquire() // P
   {s--;if (s<0) #blocage du processus dans une file#;}
  public void release // V
    {s++;if (s<=0) #réveil d'un processus#;}
   }
```

Sémaphores : exemple

```
Class tampon{
 private elt[N] le_tampon;
 private Sémaphore mutex, non_vide, non_plein;
 public tampon{
  mutex     = new Semaphore(1);
  non_vide  = new Semaphore(0);
  non_plein = new Semaphore(n);
  …
  }
Public produire(elt e){
  non_plein.aquire();
  mutex.acquire();
  this.ranger(e);
  mutex.release() ;
  non_vide.release() ;
}
```

```
Public elt consommer(){
  non_vide.aquire();
  mutex.aquire();
  e = this.sortir();
  mutex.release() ;
  non_plein.release() ;
  }
```

Sémaphores : exemple

```
Process producteur{
  tantque (non fini) {
    <calcul d'une valeur de nouv_e>;
    tampon.produire(nouv_e);
    }
}
Process consommateur{
  tantque (non fini) {
    e_frais = tampon.consommer();
    <utilisation de e_frais>;
    }
}
```

Sémaphores : exemple

- Mutex assure l'atomicité des opérations ranger et sortir
- non_vide et non_plein vérifient les préconditions de consommer et produire

Sémaphores : compléments Java

- on peut passer un paramètre à acquire et release
  ```
  void acquire(int permits)
  void release(int permits)
  ```
- on peut tester la valeur du compteur
  ```
  int availablePermits()
  ```
- on peut diminuer la valeur du compteur sans se bloquer
  ```
  protected void reducePermits()
  ```
- on peut savoir si des threads attendent et estimer combien
  ```
  boolean hasQueuedThreads()
  int getQueueLength()
  ```
- on peut faire un acquire non blocant
  ```
  boolean tryAcquire()
  ```

Sémaphores : remarques

- Intérêts
  - mise en œuvre assez facile
  - mécanisme simple à comprendre
- Inconvénients
  - signification du compteur pas toujours évidente
  - mécanisme d'assez bas niveau
  - synchronisations disséminées dans le code
  - il est facile de faire des erreurs

Régions critiques conditionnelles

- Régions critiques

```
var partagée type v;
région v faire <instructions>
```

  - les régions critiques associées à une même variable sont mutuellement exclusives
  - une variable partagée ne peut-être accédée qu'a l'intérieur d'une région critique

Régions critiques conditionnelles

- Régions critiques conditionnelles

```
région v faire
  { <bloc1> ; attendre b(v); <bloc2> }
```

  - b(v) est une expression booléenne
    - si b(v) est fausse le processus est bloqué
    - à chaque fois que v est modifiée, b(v) est réévaluée automatiquement
    - qd b(v) redevient vraie le processus suspendu peut rentrer dans sa section critique
    - système de réveil automatique

Régions critiques conditionnelles : exemple

```
Class tampon{
  boolean plein, vide;
  public void ranger(elt e){
    <mettre la valeur e dans le tampon>
    if (<le tampon est plein>) this.plein = true;
    this.vide = false;
  }

  public elt sortir(){
    <prendre la valeur dans le tampon>
    if (<le tampon est vide>) this.vide = true;
    this.plein = false;
  }
}
```

Régions critiques conditionnelles : exemple

```
Class xxx {// non implémenté en Java
  Public Produire(elt e){
    région tampon {
    attendre(tampon.plein == false);
    tampon.ranger(e);
  }
   Public elt consommer(){
    région tampon {
    attendre(tampon.vide == false);
    e = tampon.sortir();
  }
```

Régions critiques conditionnelles : exemple

```
Process producteur{
  tantque (non fini) {
    <calcul d'une valeur de nouv_e>;
    xxx.produire(nouv_e);
     }
}
Process consommateur{
  tantque (non fini) {
    e_frais = xxx.consommer();
    <utilisation de e_frais>;
     }
}
```

Régions critiques conditionnelles

- Intérêts
  - regroupe les opérations et les synchronisations
  - pas de manipulation de variables de synchronisation
- inconvénients
  - coûteux à mettre en œuvre
  - réévaluations parfois inutiles

*

Moniteurs [C.A.R Hoare xx]

- Sépare
  - opérations de blocages
    - attendre() : blocage d'un processus dans une file
    - reprendre() : libère un processus bloqué
  - gestion des variables de synchronisation
    - protégées par exclusion mutuelle
- Regroupe
  - synchronisations et accès aux variables dans un même objet

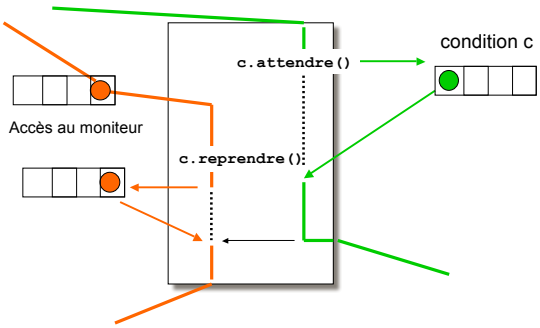
Moniteurs de HOARE

```
Class exemple implements HOARE_Monitor {//Non Java
  //les méthodes s'exécutent toutes en exclusion
  mutuelle
     private <toutes les variables>;
     private condition c1; //déclaration d'une file
     private condition c2; //déclaration d'une file
     public exemple { … };
     public m1(...){
     …
     if (…) c1.attendre();
     …
     if (…) c2.reprendre()};
     public m2(...){...};
     private m_local1(..){…}
  }
```

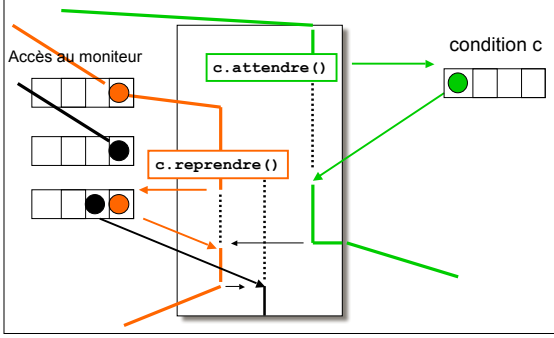
Moniteurs de HOARE : Sémantique des opérations

- Accès :
  - à un instant donné, un et un seul processus exécute des instructions d'un moniteur
- c.attendre()
  - bloque le processus dans la file "c"
  - l'accès au moniteur est libéré
- reprendre()
  - libère immédiatement un processus bloqué :le processus libéré se voit allouer le moniteur
  - le processus qui a effectué le reprendre() est bloqué

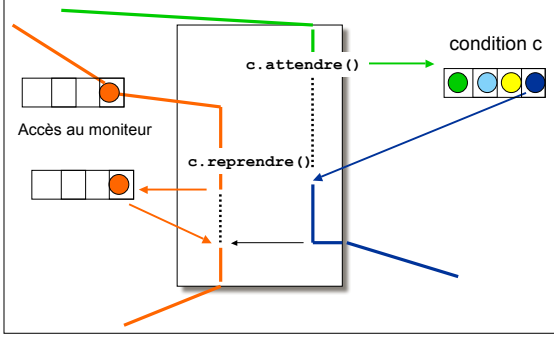
Moniteurs de HOARE : Sémantique des opérations



Moniteurs de HOARE :
Sémantique des opérations



Moniteurs de HOARE :
Sémantique des opérations



Moniteurs de HOARE:
exemple du producteur/consommateur

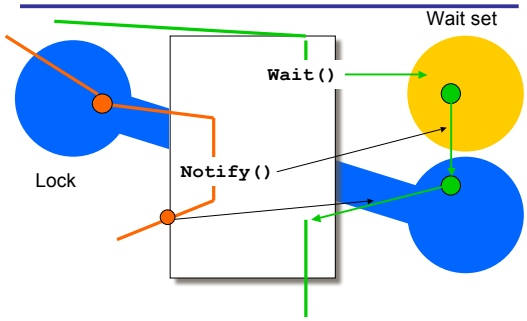
```
Class ProdCons implements HOARE_Monitor {//Non Java
  private elt[N] le_tampon; ...
  private condition pourProduire;
  private condition pourConsommer;

  public Produire(elt e){
    if (this.plein)pourProduire.attendre();
    this.ranger(e);
    pourConsommer.reprendre();
  };
  public elt Consommer(){
    if (this.vide) pourConsommer.attendre();
    e = this.sortir();
    pourProduire.reprendre();
  };
  private ranger(elt e){…}
  private elt sortir(){…}
}
```

Outils de synchronisation JAVA

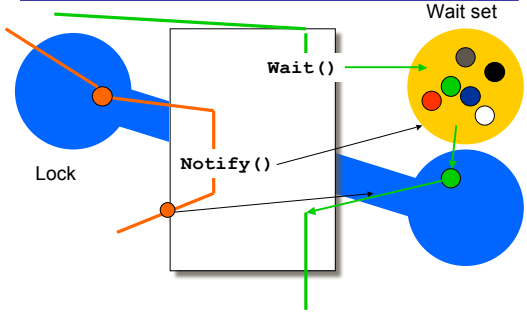
- **`Wait()  / notify() / notifyAll()`**

`wait()`	bloque le thread dans un ensemble d 'attente jusqu 'à ce qu 'il soit « notifié »
`wait(long timeout)`	… Jqa timeout
`wait(long timeout, int nanos)`	…+nanos
`notify()`	réveille un des threads bloqués dans cet ensemble
`notifyAll()`	réveille tous les threads bloqués dans cet ensemble

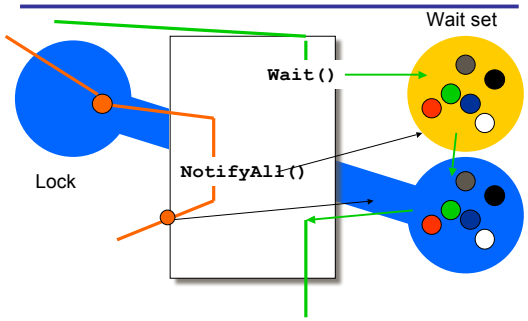
Synchronisations par blocage/réveil



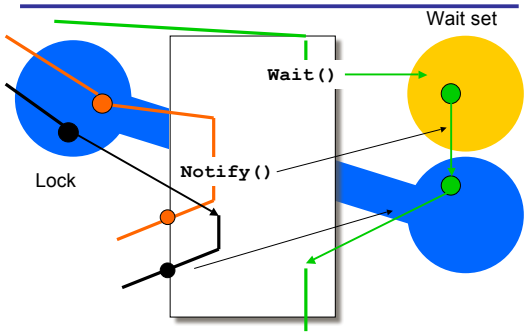
Synchronisations par blocage/réveil



Synchronisations par blocage/réveil



Synchronisations par blocage/réveil



Java
Exemple producteur / consommateur (1 case)

```
public synchronized int get() {
  while (available == false) {
     try{
        // attendre la production d'une valeur
        wait();
        } catch (InterruptedException e) {
           }
    }
  available = false;
  // prévenir que la valeur a été consommée
  notifyAll();
  return contents;
}       /* exemple du tutorial Java SUN */
```

Suite ...

```
public synchronized void put(int value) {
  while (available == true) {
      try {
          // attendre que la valeur soit consommée
          wait();
          } catch (InterruptedException e) {
             }
    }
  contents = value;
  available = true;
  // prévenir qu'une nouvelle valeur a été produite
  notifyAll();
} /* exemple du tutorial Java SUN */
```

Threads Posix : Sémaphores d 'exclusion mutuelle

- 2 types de sémaphores
  - rapides `PTHREAD_MUTEX_INITIALIZER;`
  - réentrants `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP`
- création/initialisation

```
pthread_mutex_t sema = PTHREAD_MUTEX_INITIALIZER_NP
```

- utilisation

```
pthread_mutex_lock(&sema);
< section critique >
pthread_mutex_unlock(&sema);
```

- destruction

```
rc = pthread_mutex_destroy(&sema);
```

Sémaphores d 'exclusion mutuelle

- Plus précisément:

```
int rc = pthread_mutex_lock(&sema);
if (rc) { /* erreur d 'init/signal/... */
    perror("pthread_mutex_lock");
    pthread_exit(NULL);
```

Threads Posix : Conditions (files d'attente)

- création/initialisation

```
pthread_cond_t pour_faire = PTHREAD_COND_INITIALIZER;
```

- blocage

```
rc = pthread_cond_wait(&pour_faire, &mutex);
ou
pthread_cond_timedwait(&pour_faire, &mutex, &timeout);
```

- réveil (pas besoin du sémaphore d 'exclusion mutuelle !)

```
rc = pthread_cond_signal(&pour_faire)
ou
rc = pthread_cond_broadcast(&pour_faire)
```

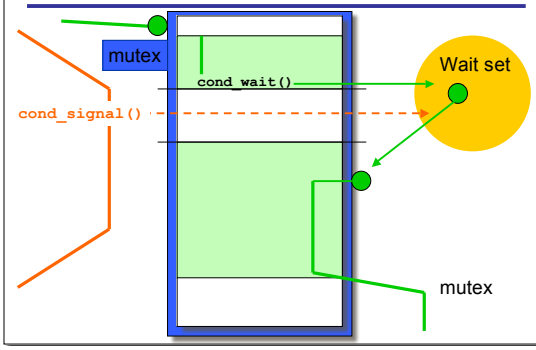
- destruction

```
rc = pthread_cond_destroy(&pour_faire);
```

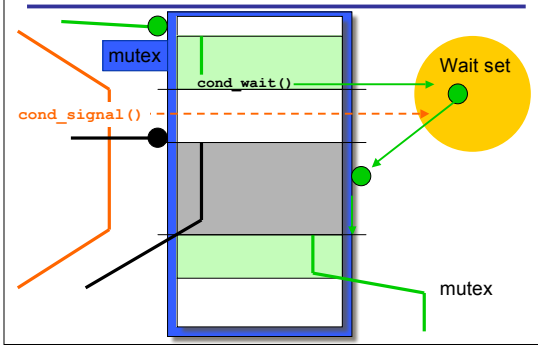
Threads Posix: Conditions (utilisation)

```
pthread_mutex_lock(&mutex);

  while (peux_pas_faire())
     pthread_cond_wait(&pour_faire, &mutex);

   <faire>

pthread_mutex_unlock(&mutex);
...
if (peux_faire())
   pthread_cond_signal(&pour_faire)
```

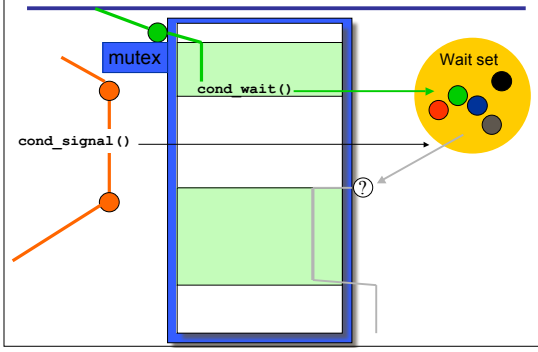
Threads Posix: Conditions



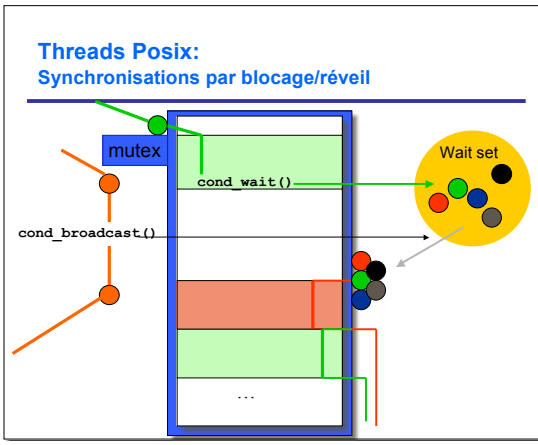
Threads Posix:
Conditions



Threads Posix:
Synchronisations par blocage/réveil



Threads Posix:
Synchronisations par blocage/réveil



Moniteurs avec conditions de Kessel

- Problème des réveils explicites
  - risque d'oublier les .reprendre()
    - blocages infinis
- Conditions de Kessel
  - Associer aux files des expression booléennes
    - Kcondition = file + expression booléenne
  - Réévaluation des conditions "automatiques"
    - lorsqu'un processus
      - sort du moniteur
      - se bloque

Moniteurs avec conditions de Kessel : exemple

```
Class ProdCons implements Kessel_Monitor {//Non Java
  private elt[N] le_tampon; ...
  private Kcondition pourProduire{nbElt < max};
  private Kcondition pourConsommer{nbElt > 0};

  public Produire(elt e){
    pourProduire.attendre();
    this.ranger(e); // doit faire la maj de nbElt

  };
  public elt Consommer(){
    pourConsommer.attendre();
    e = this.sortir(); // doit faire la maj de nbElt

  };
  private ranger(elt e){…}
  private elt sortir(){…}
}
```

Remarque

- Insuffisances des mécanismes
  - mécanismes fondés sur l'exclusion mutuelle
    - n'est pas général
    - contorsions pour l'écriture de certains programmes
  - abstraction insuffisante
    - décrit la mise en œuvre des paradigmes
    - il faudrait décrire seulement le paradigme

Lecteurs / rédacteurs

- paradigme :
  - plusieurs processus peuvent accéder à la structure de donnée en parallèle en lecture
  - 1 et 1 seul processus peut accéder à la structure de données pour réaliser une écriture
  - lecture et écriture sont mutuellement exclusives
- Mise en œuvre avec un moniteur
  - mise en œuvre directe
    - contrainte inutilement forte (exclusion mutuelle)

Lecteurs / rédacteurs: une mise en œuvre avec des moniteurs

```
//Processus réalisant une lecture :
LecRed.DebutLire();
e = journal.lire();
LecRed.FinLire();
```

```
//Processus réalisant une écriture :
LecRed.DebutEcrire();
journal.ecrire(e);
LecRed.FinEcrire();
```

Lecteurs / rédacteurs: une mise en œuvre avec des moniteurs

```
Class LecRed implements HOARE_Monitor {
   ...
  private condition pourEcrire, pourLire;
  public DebutLire(){
    if (nbRed >0) pourLire.attendre();
    nbLect++;
    pourLire.reprendre();
  };
  public FinLire(){
    nbLect--;
    if (nbLect == 0)pourEcrire.reprendre();
  };
  public DebutEcrire(){
    if ((nbRed != 0) || (nbLect != 0))
        pourEcrire.attendre();
    nbRed++;
  }
  public FinEcrire(){
    nbRed--;
    pourEcrire.reprendre();
    if (nbRed=0) pourLire.reprendre();
  }
}
```

Compteurs de synchronisation

- But
  - permettre la mise en œuvre de synchronisations moins restrictives que l'exclusion mutuelle
  - réaliser la mise à jour automatique de variables de synchronisation
  - gérer les réveils automatiquement
  - regroupement dans une même unité syntaxique de la synchronisation (module de contrôle)
  - séparation synchronisation / opérations

Compteurs de synchronisation

- Description
  - les synchronisations portent sur l'état de compteurs
  - contrôle de l'accès aux méthodes d'un ou plusieurs objets différents par une expression sur ces compteurs
  - les accès aux méthodes sont automatiquement comptabilisés

Compteurs de synchronisation

`m.req:` nombre d'invocations de m réalisés depuis le début de l'exécution du programme. Ces invocations peuvent ou non avoir été autorisés à être réalisées (*requêtes*)

`m.aut:` nombre d'invocations de m *autorisées* depuis le début de l'exécution du programme. Ces invocations ont été effectuées ou sont en cours d'exécution

`m.term:` nombre d'invocations de m qui ont été réalisées depuis le début de l'exécution du programme. Ces invocations sont *terminées*

- ces 3 compteurs sont croissants et suffisent pour la définition d'un grand nombre de mécanismes de synchronisation

on a toujours : `m.req>= m.aut >= m.term`

Compteurs de synchronisation

`m.att:` nombre d'invocations de m qui sont actuellement bloquées (en *attente*)

`m.act:` nombre d'invocations de m qui sont en cours actuellement (*actives*)

- ces 2 compteurs sont pratiques mais peuvent aussi se calculer à partir des 3 précédents :

```
m.att = m.req - m.aut
m.act = m.aut - m.term
```

Compteurs de synchronisation : Shéma d'exécution

<invocation de m>

- mise à jour de m.req et m.att +**R**éévaluation
- évaluation de la condition d'accès a m

<blocage éventuel>

- mise à jour de m.aut et m.att + **R**

<exécution du corps de m>

- mise à jour de m.term et m.act + **R**

<retour>

Compteurs de synchronisation : Exemple des lecteurs/rédacteurs

```
Class LecRecSimple implements MdC {

   private condition lire{
    (ecrire.act = 0)};
  private condition ecrire{
    (ecrire.act = 0 ) && lire.act = 0)};

  public elt lire(){…}
  public ecrire(elt e){…};
}
```

```
Class LecRecPri implements MdC {

   private condition lire{
    ((ecrire.act = 0) && (ecrire.att = 0))};
  private condition ecrire{
    (ecrire.act = 0 ) && lire.act = 0)};
     …
}
```

Compteurs de synchronisation : Exemple du producteur/consommateur

```
Class ProdCons implements MdC {

   private condition produire{
    (produire.aut - consommer.term < N)};

  private condition consommer{
    (produire.term - consommer.aut > 0)};

  public elt consommer(){…}
  public produire(elt e){…};
}
```

Compteurs de synchronisation

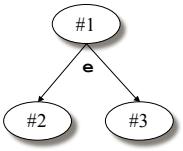
- Intérêt
  - spécification plus abstraite
  - concis
- Inconvénients
  - difficile à mettre en œuvre
  - réévaluations des conditions parfois inutiles
- Utilisation
  - pour spécifier avant d'utiliser d'autres mécanismes

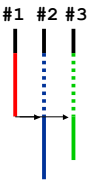
Evénements/ futurs

- Définition des événements
  - variable à assignation unique
  - un seul processus peut l'initialiser
  - tout processus peut la lire
  - lecture blocante jqa ce que la variable ait reçu une valeur
- Utilisation
  - coopération simple entre processus

Evénements/ futurs : exemple

```
Calcul de
V1 = f(a,b) + c * d
et
V2 = f(a,b) +f
```





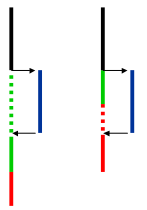

```
Event int e;

process #1 :
int a, b;
…
e = f(a,b);

process #2 :
int V1, c, d;
…
V1 = e + c * d;

process #3 :
int V2, f;
…
V2 = e + f
```

Futurs

- Même concept pour les objets
  - appel de méthode asynchrone :
    - on lance un Thread dont le résultat est une référence sur un *futur*
    - tout accès au *futur* est blocant tant que le Thread n'a pas terminé son exécution

Futurs

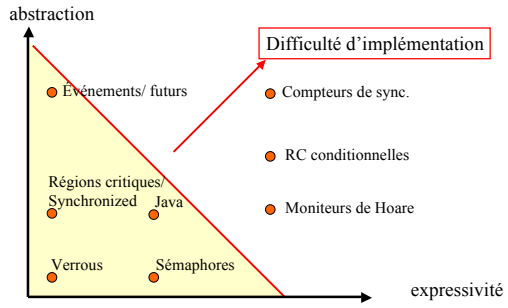
classique | futurs



```
. . .
f = MyObject.m();
. . .
System.out.println(f.x+f.y)
```

```
. . .
Futur f = MyObject.m();
. . .
System.out.println(f.x+f.y)
```

Comparaison des mécanismes de synchronisation



Conclusion

- Simple ?
  - Mécanismes de synchronisation
    - puissants et abstraits
      OU
    - disponibles
  - Paradigmes de synchronisation
    - faciles à énoncer
      ET
    - difficile à respecter

Conclusion

- Conseils
  - Ne pas confondre
    - paradigme et mécanisme
  - Ne pas réinventer la roue
    - d'abord spécifier le problème
    - se référer à des paradigmes connus

Les synchronisations les meilleures sont les plus simples