Moniteurs, Java, Threads et Processus

Une vue orientée-objet de la mémoire partagée

- On peut voir un sémaphore comme un objet partagé accessible par deux méthodes : `wait` et `signal`.

- L'idée du concept de moniteur est de généraliser cela à des objets et méthodes quelconques.

- Si plusieurs processus peuvent exécuter des méthodes sur un même objet, le problème de gérer l'interaction entre ces processus se pose :

  - Il faut assurer une certaine atomicité garantissant l'exécution correcte des opérations ;

  - Un mécanisme permettant la mise en attente de processus par rapport à une condition sur l'état de l'objet partagé est nécessaire.

Moniteurs : concept

- Un *moniteur* est une classe utilisée dans un contexte de parallélisme. Les instances de la classe seront donc des objets utilisés simultanément par plusieurs processus.

- Définir un moniteur se fait en définissant la classe correspondante. Nous utiliserons une syntaxe inspirée de Java.

Un exemple de classe : stack

```
public class Stack {
  protected static int max = 300;
  private int nbElements;
  private Object[] contenu;

public Stack()
  { nbElements = 0;
    contenu = new Object[max];
  }
```

```
public void push(Object e)
{
  if (nbElements < max)
    contenu[nbElements++] = e;
}
```

```
public Object pop()
{
  if (nbElements > 0)
     return contenu[--nbElements];
  else
    return null;
}
```

```
}
```

Moniteurs : première règle de synchronisation

- Même pour une classe aussi simple que `Stack`, l'exécution simultanée par des processus différents de méthodes de la classe peut poser problème.

- Une première règle de synchronisation imposée dans le cadre des moniteurs est donc la suivante :

  Les méthodes d'un moniteur sont exécutées en exclusion mutuelle.

Note Java. En Java, l'exclusion mutuelle des méthodes doit se spécifier de façon explicite à l'aide de l'attribut `synchronized`.

Le problème du producteur et consommateur dans le cadre des moniteurs

Ce problème se résout naturellement à l'aide de la classe suivante.

```
public class PCbuffer
{
  private Object buffer[];      /* Mémoire partagée    */
  private int N;                /* Capacité de la zone */
  private int count, in, out;   /* nb d'éléments, pointeurs */

  public PCbuffer(int argSize)
   {                            /* création d'un tampon de taille
                                   argSize */
    N = argSize;
    buffer = new Object[N];
    count = 0; in = 0; out = 0;
  }
```

```
  public synchronized void append(Object data)
  {
    buffer[in] = data;
    in = (in + 1) % N; count++;
  }

  public synchronized Object take()
  {
    Object data;

    data = buffer[out];
    out = (out + 1) % N;
    count--; return data;
  }
}
```

Il manque évidemment la synchronisation nécessaire lorsque la zone tampon est pleine ou vide.

Moniteurs : la synchronisation par files d'attente

- Pour permettre la mise en attente de processus lorsqu'ils ne peuvent exécuter l'opération souhaitée, on utilise des files d'attente.

- Ces files d'attente sont gérées par trois opérations.
  - `qWait()` : met le processus exécutant cette opération en attente dans la file.
  - `qSignal()` : débloque le premier processus en attente dans la file (sans effet si la file est vide).
  - `qNonempty()` : teste que la file n'est pas vide.

- Lorsqu'un processus est mis en attente par une opération `qWait()`, il libère l'exclusion mutuelle liée à l'exécution d'une méthode du moniteur.

Les files d'attente vues comme une classe

Les files d'attente seront vues comme des objets instanciés à partir de la classe suivante dont la concrétisation sera donnée par la suite.

```
public class Waitqueue
{     ....              /* données à préciser */
   public Waitqueue()
   {  ....              /* constructeur à préciser */
   }
   public void qWait()
   {  ....              /* opération de mise en attente */
   }
   public void qSignal()
   {  ....              /* opération d'activation  */
   }
   public boolean qNonempty()
   { ....               /* test de l'état non vide de la file */
   }
}
```

Le problème du producteur et consommateur avec files d'attente

Pour synchroniser les producteurs et consommateurs, on utilise deux files d'attente : une pour les processus en attente d'un élément à consommer, une pour les processus en attente d'une place dans le tampon.

```
public class PCbuffer
{
  private Object buffer[];     /* Mémoire partagée    */
  private int N ;              /* Capacité de la zone */
  private int count, in, out;  /* nb d'éléments, pointeurs */
  private Waitqueue notfull, notempty; /* files d'attente */

  public PCbuffer(int argSize)
    { N = argSize;              /* création d'un tampon de taille
      buffer = new Object[N];      argSize */
      count = 0; in = 0; out = 0;
      notfull = new Waitqueue(); notempty = new Waitqueue();
    }
```

```
  public synchronized void append(Object data)
  { if (count == N) notfull.qWait();
    buffer[in] = data;
    in = (in + 1) % N; count++;
    notempty.qSignal();
  }

  public synchronized Object take()
  { Object data;

    if (count == 0) notempty.qWait();
    data = buffer[out];
    out = (out + 1) % N;
    count--; notfull.qSignal();
    return data;
  }
}
```

L'opération `qSignal` n'a pas d'effet si la file est vide et donc cette solution ne souffre pas du problème de double signalisation rencontré dans le cadre de l'utilisation des sémaphores binaires.

Les priorités lors d'une opération `qSignal`

- Lorsque l'on exécute une opération `qSignal`, le processus en tête de la file d'attente est débloqué, mais *a priori* cela ne signifie pas qu'il poursuit son exécution.

- Cela pose potentiellement un problème, car la situation qui a provoqué le `qSignal` (par exemple zone tampon non vide) pourrait ne plus être vraie lorsque le processus débloqué reprend effectivement son exécution (par exemple, parce qu'un autre processus a pris le seul élément restant dans une zone non vide).

- Pour éviter cela, on impose la règle suivante sur la synchronisation des moniteurs :

  Reprise immédiate : Le processus débloqué lors d'une opération `qSignal` a priorité sur tout autre processus tentant d'exécuter une opération sur l'objet auquel la file d'attente est attachée.

- Qu'en est-il du processus exécutant l'opération `qSignal` ?

  – Le processus exécutant l'opération `qSignal` a priorité pour terminer la méthode en cours d'exécution, lorsque le processus débloqué a terminé son opération sur le moniteur.

Les sémaphores vus comme moniteurs

Avec le mécanisme de file d'attente, les sémaphores peuvent être facilement implémentés comme des moniteurs. Cela permet aussi de prévoir d'autres opérations sur un sémaphore, en particulier le test du nombre de processus en attente sur le sémaphore, ce qui nous sera utile par la suite.

```
public class Semaphore
{ int value;                  /* la valeur du sémaphore */
  int nbWait = 0;             /* le nombre de processus
                                 en attente */
  Waitqueue queue;

  public Semaphore(int inival)
  {  value = inival;
     queue = new Waitqueue();
  }
```

```
  public synchronized void semWait()
  {  if (value <= 0)
     {  nbWait++;
        queue.qWait();
        nbWait--;
     }
     value--;
  }

  public synchronized void semSignal()
  {  value++; queue.qSignal();
  }

public synchronized int semNbWait()
  {  return nbWait;
  }
```

Remarquer la nécessité de la reprise immédiate pour que cette implémentation soit correcte.

L'implémentation des files d'attente

- Il est clair que l'implémentation des files d'attente nécessite un mécanisme de base de mise en attente des processus qui doit être fourni au niveau de l'environnement d'exécution du programme.

- Nous allons temporairement ignorer ce problème en supposant que nous disposons de sémaphores implémentés directement, sans utiliser les files comme présenté ci-dessus.

- La reprise immédiate pose un problème, puisqu'elle crée un lien entre la gestion des files et l'accès en exclusion mutuelle aux méthodes du moniteur. Nous examinerons d'abord une première implémentation, sans reprise immédiate, avant d'introduire cette contrainte dans un deuxième temps.

Une implémentation des files sans reprise immédiate

Une file est simplement implémentée à l'aide d'un sémaphore (supposé FIFO) dont la valeur est toujours 0.

```
public class Waitqueue
{  private int qcount = 0;  /* le nombre de processus en attente */
   private Semaphore Qsem;

   public Waitqueue()
   {  Qsem = new Semaphore(0);
                          /* création du sémaphore initialisé à 0 */
   }

   public void synchronized qWait()  /* opération de mise en attente */
   {  qcount++; Qsem.semWait(); qcount--;
   }
```

```
   public void synchronized qSignal()  /* opération d'activation  */
   {  if (qcount > 0) Qsem.semSignal();
   }

public boolean synchronized qNonempty()
   {  return qcount == 0;     /* test de l'état non vide de la file */
   }
}
```

Indépendamment du problème de la reprise immédiate, cette solution souffre d'un risque de deadlock lié à une multiplication des verrouillages par `synchronized`.

Note Java : `synchronized` et les verrous

- Une méthode déclarée **`synchronized`** est exécutée en exclusion mutuelle par rapport aux autres méthodes relatives au même objet.

- Cela peut se comprendre en disant qu'il y a un *verrou* (qui pourrait par exemple être implémenté par un sémaphore) lié à l'objet.

- Quand une méthode **`synchronized`** appelle une méthode **`synchronized`** d'un autre objet, un deuxième verrouillage sur ce nouvel objet est donc effectué.

- Le problème arrive quand un processus est mis en attente. Dans ce cas, il faut libérer le verrouillage, mais cela est difficile à réaliser s'il y a eu une série d'opérations de verrouillage.

- En Java, l'opération élémentaire de mise en attente (voir plus loin) ne libère que le verrou de l'objet courant.

Une implémentation des files avec reprise immédiate

- Pour résoudre à la fois le problème de la reprise immédiate et le problème du verrouillage multiple, nous allons gérer explicitement l'exclusion mutuelle entre méthodes d'un moniteur à l'aide d'un sémaphore.

- Quand un processus sera mis an attente, l'opération `qWait` libérera explicitement l'exclusion mutuelle associée à la méthode appelant l'opération `qWait`.

- Pour ce faire, un argument indiquant le sémaphore d'exclusion mutuelle à libérer sera donné à `qWait` (ainsi qu'à `qSignal` pour pouvoir gérer la reprise immédiate).

Une implémentation des files avec reprise immédiate (suite)

```
public class Waitqueue
{  private int qcount = 0;  /* le nombre de processus en attente */
   private Semaphore Qsem;

   public Waitqueue()
   {  Qsem = new Semaphore(0);
                          /* création du sémaphore initialisé à 0 */
   }

   public void  qWait(Semaphore Mutsem)
                         /* opération de mise en attente */
   {  qcount++; Mutsem.semSignal(); Qsem.semWait(); qcount--;
   }
```

```
   public void  qSignal(Semaphore Mutsem)
                              /* opération d'activation  */
   {  if (qcount > 0) {
        Qsem.semSignal(); Mutsem.semWait();
      }
   }

public boolean  qNonempty()
   {  return qcount == 0;   /* test de l'état non vide de la file */
   }
}
```

Pour éviter les blocages, l'exclusion mutuelle n'est pas imposée sur les méthodes de manipulation de files d'attente. Les interférences non désirées son toutefois exclues, vu qu'une file est utilisée par un seul objet partagé et que les méthodes de cet objet sont déjà exécutées en exclusion mutuelle.

La zone tampon utilisant l'implémentation des files par sémaphores

```
public class PCbuffer
{
  private Object buffer[];     /* Mémoire partagée    */
  private int N ;              /* Capacité de la zone */
  private int count, in, out;  /* nb d'éléments, pointeurs */
  private Waitqueue notfull, notempty; /* files d'attente */
  private mutex Semaphore;     /* sémaphore d'exclusion mutuelle */

  public PCbuffer(int argSize)
  {  N = argSize;
     buffer = new Object[N];
     count = 0; in = 0; out = 0;
     notfull = new Waitqueue(); notempty = new Waitqueue();
     mutex = new Semaphore(1);
  }
```

```
  public  void append(Object data)
  { mutex.semWait()
    if (count == N) notfull.qWait(mutex);
    buffer[in] = data;
    in = (in + 1) % N; count++;
    notempty.qSignal(mutex);
    mutex.semSignal();
  }

  public Object take()
  { Object data;

    mutex.semWait()
    if (count == 0) notempty.qWait(mutex);
    data = buffer[out];
    out = (out + 1) % N;
    count--;
    notfull.qSignal(mutex);
    mutex.semSignal(); return data;
  }
}
```

L'urgence des processus ayant exécuté une opération
qSignal

- Dans l'implémentation précédente, la priorité n'a pas été donnée aux processus ayant effectué une opération `qSignal`.
- Pour ce faire, un deuxième sémaphore d'exlusion mutuelle `urgent` sera utilisé.
- L'opération `qWait` sera appelée avec le sémaphore d'exclusion mutuelle ou le sémaphore `urgent` suivant qu'il y a ou non des processus en attente sur ce dernier.
- Pour ce faire on comptera explicitement le nombre de processus en attente sur `urgent`. En effet, un appel à `semNbWait(urgent)` ne compte pas les processus qui ont exécuté une opération `qSignal`, vont se mettre en attente sur `urgent`, mais ne l'ont pas encore fait.
- En fin de méthode, on exécute une opération `semSignal` sur `urgent` s'il y a un processus en attente sur ce sémaphore ; si non, on exécute l'opération sur le sémaphore d'exclusion mutuelle.

Une implémentation des files avec reprise immédiate et priorité aux processus signalant)

```
public class PCbuffer
{
  private Object buffer[];      /* Mémoire partagée    */
  private int N ;               /* Capacité de la zone */
  private int count, in, out;   /* nb d'éléments, pointeurs */
  private Waitqueue notfull, notempty; /* files d'attente */
  private Semaphore mutex, urgent;     /* sémaphores */
  private int urcount;          /* nb de processus en attente
                                   sur urgent */


  public PCbuffer(int argSize)
  { N = argSize;
    buffer = new Object[N];
    count = 0; in = 0; out = 0; urcount = 0;
    notfull = new Waitqueue(); notempty = new Waitqueue();
    mutex = new Semaphore(1); urgent = new Semaphore(0);
  }
```

```
public  void append(Object data)
{ mutex.semWait()
  if (count == N) {
     if (urcount > 0) notfull.qWait(urgent);
       else notfull.qWait(mutex);
    }
  buffer[in] = data;
  in = (in + 1) % N; count++;
  urcount++;
  notempty.qSignal(urgent);
  urcount--;
  if (urcount > 0) urgent.semSignal();
     else mutex.semSignal();
}
```

```
  public  Object take()
  { Object data;

    mutex.semWait()
     if (count == 0) {
       if (urcount > 0) notempty.qWait(urgent);
         else notempty.qWait(mutex);
      }
    data = buffer[out];
    out = (out + 1) % N;
    count--;
    urcount++;
    nofull.qSignal(urgent);
    urcount--;
    if (urcount > 0) urgent.semSignal();
       else mutex.semSignal();
    return data;
  }
}
```

Note Java : l'implémentation des sémaphores

- Pour éviter la circularité dans l'implémentation des moniteurs, il faut une implémentation des sémaphores qui ne fasse pas appel aux files d'attente définies.

- Java fournit les primitives permettant de réaliser une telle implémentation.

  – D'une part, le caractère `synchronized` d'une méthode en garantit l'exécution en exclusion mutuelle.

  – D'autre part, Java permet la mise en attente de processus grâce aux méthodes `wait()`, `notify()` et `notifyAll()`.

Note Java : `wait` et `notify`

- `wait()` : Suspend le processus courant et libère l'exclusion mutuelle relative à l'objet courant.

- `notify()` : Sélectionne un processus en attente sur l'objet courant et le rend exécutable. Le processus sélectionné doit réacquérir l'exclusion mutuelle associée à l'objet (il n'y a pas de reprise immédiate).

- `notifyAll()` : Similaire à `notify()`, mais rend exécutable tous les processus en attente sur l'objet courant.

Une première implémentation des sémaphores en Java

```
public class Semaphore
{ private int value;                    /* la valeur du sémaphore */
  private int nbWait = 0;               /* nb en attente */

public Semaphore(int inival)
  {  value = inival;
  }

  public synchronized void semWait()
  {  while (value <= 0)
     {  nbWait++;
        wait();
     }
     value--;
  }
```

Vu l'absence de reprise immédiate, la valeur du sémaphore doit être testée à nouveau lorsque l'on sort du `wait()`.

```
  public synchronized void semSignal()
  {  value++;
     if (nbWait > 0)
       {  nbWait--; notify();
       }
  }

  public synchronized int semNbWait()
  {  return nbWait;
  }
}
```

L'opération `nbWait--` se fait dans `semSignal()` juste avant le `notify()` car, étant donné l'absence de reprise immédiate, effectuer cette opération dans `semWait()` après le `wait()` pourrait mener au renvoi d'une valeur incorrecte par `semNbWait()`.

La sémantique de `wait()` et `notify()` implique que cette implémentation n'est pas équitable.

Une implémentation équitable des sémaphores en Java

- Pour obtenir une implémentation équitable, il faut explicitement implémenter une file d'attente.

- La file d'attente sera une file d'objets sur chacun desquels exactement un processus sera mis en attente.

- L'absence d'équité lors de l'exécution d'une opération `notify()` ne posera alors plus de problème.

Une implémentation équitable des sémaphores en Java (suite)

La classe `Queue` implémente une file d'attente classique.

```
public class SemaphoreFIFO
{ private int value;                    /* la valeur du sémaphore */
  private int nbWait = 0;               /* nb en attente */
  private Queue theQueue;               /* la file d'attente explicite */

  public SemaphoreFIFO(int inival)
  {  value = inival;
     theQueue = new Queue();
  }
```

```
public  void semWait()
{  Semaphore semElem;
   synchronized(this){
     if (value == 0)
       { semElem  = new Semaphore(0);
         theQueue.enqueue(semElem);
         nbWait++;
       }
       else
         {
           semElem  = new Semaphore(1);
           value--;
         }
   } /* synchronized */

   semElem.semWait();
}
```

On sort l'opération `semElem.semWait()` de l'exclusion mutuelle pour éviter un double verrouillage. Remarquer l'initialisation différente de `semElem` suivant que `value` ait la valeur `0` ou non.

```
  public synchronized void semSignal()
  {  Semaphore semElem;
     if (!theQueue.empty()){
        semElem = theQueue.dequeue();
        nbWait--; semElem.semSignal();
     }
       else value++;
  }

  public synchronized int semNbWait()
  { return nbWait;
  }
}
```

L'opération `semSignal()` débloque le processus bloqué sur le premier élément de la file, s'il y en a un. Si non, la valeur de `value` est incrémentée.

Processus et Threads

Dans le langage Java on parle de *Threads* et non de processus. Quelle est la différence.

- Un "thread" correspond à une exécution ayant son propre flux de contrôle ("thread" peut se traduire par "fil").

- On peut implémenter les "threads" à l'aide de processus, mais cela présente quelques inconvénients.

  - Les processus opèrent dans des espaces virtuels distincts. Le partage d'objets entre "threads" est donc relativement complexe à organiser.

  - Vu la séparation forte entre le contexte des processus, le passage d'un processus à un autre est une opération relativement lourde et donc lente.

Processus et Threads (suite)

- Plusieurs systèmes modernes ont une notion de “thread” ou “processus léger”. Ces processus se distinguent des processus traditionnels étant conçus pour coopérer (en partageant par exemple un espace virtuel) plutôt que pour travailler dans des contextes totalement distincts.

  – Les “threads” système sont apparus comme outil d’exploitation des machines parallèles.

  – Les “threads” Java peuvent être implémentés par des “threads” systèmes.

- On peut aussi implémenter les “threads” Java à l’intérieur d’un processus système en incorporant dans le code exécuté un gestionnaire de tâches simple.