5

Processus, threads et gestion de la synchronisation

Nous exposons ici les notions fondamentales que sont les processus et les threads, dans l'architecture des systèmes d'exploitation de type Windows NT/2000/XP. Il faut avoir à l'esprit que le CLR (ou moteur d'exécution) décrit dans le chapitre précédent est une couche logicielle chargée par l'hôte du moteur d'exécution lorsqu'un assemblage .NET est lancé.

Introduction

Un *processus* (*process* en anglais) est concrètement une zone mémoire contenant des ressources. Les processus permettent au système d'exploitation de répartir son travail en plusieurs *unités fonctionnelles.*

Un processus possède une ou plusieurs *unités d'exécution* appelée(s) *threads*. Un processus possède aussi un espace d'adressage virtuel privé accessible en lecture et écriture, seulement par ses propres threads.

Dans le cas des programmes .NET, un processus contient aussi dans son espace mémoire la couche logicielle appelée CLR ou moteur d'exécution. La description du CLR fait l'objet du chapitre précédent. Cette couche logicielle est chargée dès la création du processus par l'hôte du moteur d'exécution (ceci est décrit page 79).

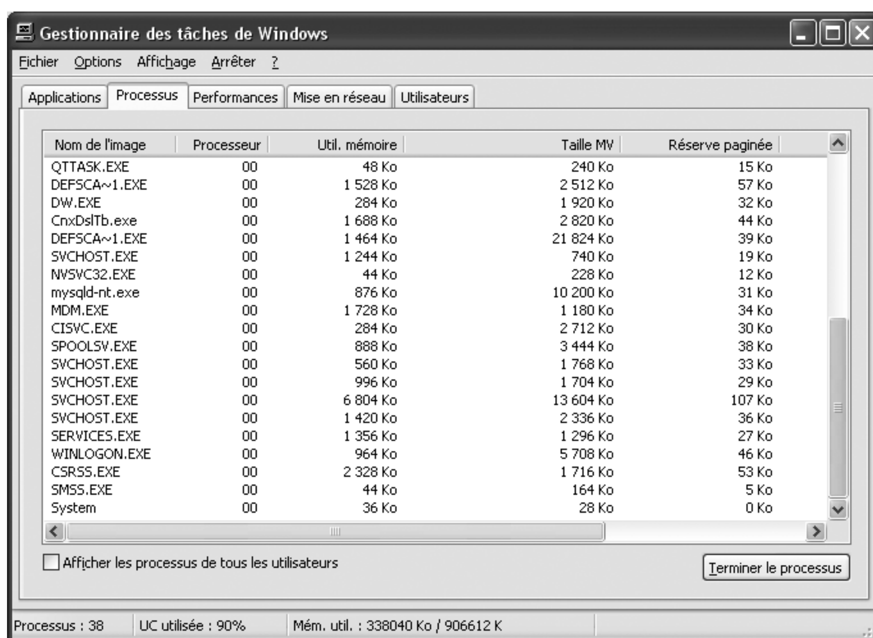
Un thread ne peut appartenir qu'à un processus et ne peut utiliser que les ressources de ce processus. Quand un processus commence, le système d'exploitation lui associe automatiquement un thread appelé *thread principal* (*main thread* ou *primary thread* en anglais). C'est ce thread qui exécute l'hôte du moteur d'exécution, le chargeur du CLR.

CS_doNet/05 Page 118 Mercredi, 14. mai 2003 10:32 10

Une *application* est constituée d'un ou plusieurs processus coopérants. Par exemple l'environnement de développement Visual Studio .NET est une application, qui peut utiliser un processus pour éditer les fichiers sources et un processus pour la compilation.

Sous les systèmes d'exploitation Windows NT/2000/XP, on peut visualiser à un instant donné toutes les applications et tous les processus en lançant le *gestionnaire des tâches* (*task manager* en anglais). Il est courant d'avoir une trentaine de processus en même temps, même si vous avez ouvert un petit nombre d'applications. En fait le système exécute un grand nombre de processus, un pour la gestion de la session courante, un pour la barre des tâches, et bien d'autres encore.

Figure 5-1. Vue des processus avec le gestionnaire des tâches



Les processus

Introduction

Dans un système d'exploitation Windows 32 bits, tournant sur un processeur 32 bits, un processus peut être vu comme un espace linéaire mémoire de 4Go (2^32 octets), de l'adresse 0x00000000 à 0xFFFFFFFF. Cet espace de mémoire est dit privé, car inaccessible par les autres processus. Cet espace se partage en 2Go pour le système et 2Go pour l'utilisateur. Windows et certains processeurs s'occupent de faire l'opération de translation entre cet espace d'adressage virtuel et l'espace d'adressage réel.

Si N processus tournent sur une machine il n'est (heureusement) pas nécessaire d'avoir Nx4Go de RAM.

CS_doNet/05 Page 119 Mercredi, 14. mai 2003 10:32 10

- Windows alloue seulement la mémoire nécessaire au processus, 4Go étant la limite supérieure dans un environnement 32 bits.
- Un mécanisme de *mémoire virtuelle* du système sauve sur le disque dur et charge en RAM des ‘morceaux’ de processus appelés pages mémoire (chaque page a une taille de 4Ko). Là encore tout ceci est transparent pour le développeur et l’utilisateur.

La classe System.Diagnostics.Process

Une instance de la classe `System.Diagnostics.Process` référence un processus. Les processus qui peuvent être référencés sont :

- Le processus courant dans lequel l’instance est utilisée.
- Un processus sur la même machine autre que le processus courant.
- Un processus sur une machine distante.

Les méthodes et champs de cette classe permettent de créer, détruire, manipuler ou obtenir des informations sur ces processus. Nous exposons ici quelques techniques courantes d’utilisation de cette classe.

Créer et détruire un processus fils

Le petit programme suivant crée un nouveau processus, appelé *processus fils*. Dans ce cas le processus initial est appelé *processus parent*. Ce processus fils exécute le bloc note. Le thread du processus parent attend une seconde avant de tuer le processus fils. Le programme a donc pour effet d’ouvrir et de fermer le bloc note :

Exemple 5-1.

```
using System.Diagnostics;
using System.Threading;
namespace ProcessTest1
{
  class Prog
  {
    static void Main(string[] args)
    {
      // crée un processus fils qui lance notepad.exe
      // avec le fichier texte hello.txt
      Process p = Process.Start("notepad.exe","hello.txt");
      // endors le thread 1 seconde
      Thread.Sleep(1000);
      // tue le processus fils
      p.Kill();
    }
  }
}
```

Notez que la méthode statique `Start()` peut utiliser les associations qui existe sur un système d’exploitation, entre un programme et une extension de fichier. Concrètement ce programme a le même comportement, si l’on écrit :

```
Process p = Process.Start("hello.txt");
```

CS_doNet/05 Page 120 Mercredi, 14. mai 2003 10:32 10

Empêcher de lancer deux fois le même programme sur la même machine

Cette fonctionnalité est requise par de nombreuses applications. En effet, il est courant qu'il n'y a pas de sens à lancer simultanément plusieurs fois une même application sur la même machine. Par exemple il n'y a pas de sens à lancer plusieurs fois *WindowMessenger* sur la même machine.

Jusqu'ici, pour satisfaire cette contrainte sous Windows, les développeurs utilisaient le plus souvent la technique dite du '*mutex nommé*' décrite page 138. L'utilisation de cette technique pour satisfaire cette contrainte souffre des défauts suivants :

- Il y a le risque faible, mais potentiel, que le nom du mutex soit utilisé par une autre application, auquel cas cette technique ne marche absolument plus et peut provoquer des bugs difficiles à détecter.
- Cette technique ne peut résoudre le cas général où l'on n'autorise que N instances de l'application.

Grâce aux méthodes statiques `GetCurrentProcess()` (qui retourne le processus courant) et `GetProcesses()` (qui retourne tous les processus lancés sur la machine) de la classe `System.Diagnostics.Process`, ce problème trouve une solution élégante et très facile à implémenter comme le prouve le programme suivant :

Exemple 5-2.

```
using System;
using System.Diagnostics;
namespace ProcessTest2
{
  class Prog
  {
    static void Main(string[] args)
    {
      if( TestSiDejaLancé() )
      {
        Console.WriteLine("Ce programme est déjà lancé.");
      }
      else
      {
        // ici le code de l'application
      }
    }
    static bool TestSiDejaLancé()
    {
      Process pcur = Process.GetCurrentProcess();
      Process[] ps = Process.GetProcesses();
      foreach( Process p in ps )
        if( pcur.Id != p.Id )
          if(pcur.ProcessName == p.ProcessName )
            return true;
```

CS_doNet/05 Page 121 Mercredi, 14. mai 2003 10:32 10


```
      return false;
    }
  }
}
```

Notez que la méthode `GetProcesses()` peut aussi retourner tous les processus sur une machine distante en communiquant comme argument le nom de la machine distante.

Les threads

Introduction

Un thread comprend :

- Un compteur d'instructions, qui pointe vers l'instruction en cours d'exécution ;
- Une pile (décrite page 100) ;
- Un ensemble de valeurs pour les registres, définissant une partie de l'état du processeur exécutant le thread ;
- Une zone privée de données.

Tous ces éléments sont rassemblés sous le nom de *contexte d'exécution du thread*. L'espace d'adressage et, par conséquent, toutes les ressources qui y sont stockées, sont communs à tous les threads d'un même processus.

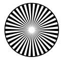
Nous ne parlerons pas de l'exécution des threads en *mode noyau* et en *mode utilisateur*. Ces deux modes, utilisés par Windows depuis bien avant .NET, existent toujours puisqu'ils se situent dans une couche en dessous du CLR. Néanmoins ces modes ne sont absolument pas visibles du Framework .NET.

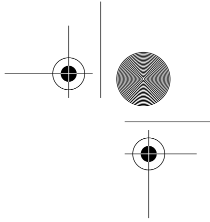
L'utilisation en parallèle de plusieurs threads constitue souvent une réponse naturelle à l'implémentation des algorithmes. En effet, les algorithmes utilisés dans les logiciels sont souvent constitués de tâches dont les exécutions peuvent se faire en parallèle. Attention, utiliser une grande quantité de threads génère beaucoup de *context switching*, et finalement nuit aux performances.

Notion de thread géré

Il faut bien comprendre que les threads qui exécutent les applications .NET sont bien ceux de Windows. Cependant on dit qu'un thread est géré quand le CLR connaît ce dernier. Concrètement, un thread est géré s'il est créé par du code géré. Si le thread est créé par du code non géré, alors il n'est pas géré. Cependant ce thread devient géré dès qu'il exécute du code géré (voir page 247 pour un exemple de code non géré qui appelle du code géré).

Un thread géré se distingue d'un thread non géré par le fait que le CLR crée une instance de la classe `System.Threading.Thread` pour le représenter et le manipuler. En interne, le CLR garde une liste des threads gérés, appelée `ThreadStore`.



CS_doNet/05 Page 122 Mercredi, 14. mai 2003 10:32 10


Le CLR fait en sorte que chaque thread géré créé soit exécuté au sein d'un domaine d'application, à un instant donné. Cependant un thread n'est absolument pas cantonné à un domaine d'application, et il peut en changer au cours du temps. La notion de domaine d'application est présentée page 15.

Notez qu'en appliquant l'attribut `ThreadStaticAttribute` à un champ statique d'une classe, vous spécifiez au CLR que ce champ n'est pas partagé entre plusieurs threads. Concrètement, chaque thread aura sa version de ces champs. La notion d'attribut est présentée page 187.

Dans le domaine de la sécurité, l'utilisateur principal d'un thread géré est indépendant de l'utilisateur principal du thread non géré sous-jacent. Tout ceci est détaillé page 180.

Le multitâche préemptif

On peut se poser la question suivante : mon ordinateur a un processeur (voir deux) et pourtant le gestionnaire des tâches indique qu'une centaine de threads tournent simultanément sur ma machine ! Comment cela est-il possible ?

Cela est possible grâce au *multitâche préemptif* qui gère *l'ordonnancement des threads*. Une partie du noyau de Windows, appelée *répartiteur* (*scheduler* en anglais), segmente le temps en portions appelées *quantum* (appelées aussi *time slices*). Ces intervalles de temps sont de l'ordre de quelques millisecondes et ne sont pas de durée constante. Pour chaque processeur, chaque quantum est alloué à un seul thread. La succession très rapide des threads donne l'illusion à l'utilisateur que les threads s'exécutent simultanément. On appelle '*contexte switching*' l'intervalle entre deux quantums consécutifs. Un avantage de cette méthode est que les threads en attente d'une ressource n'ont pas d'intervalle de temps alloué jusqu'à la disponibilité de la ressource.

L'adjectif 'préemptif' utilisé pour une telle gestion du multitâche vient du fait que les threads sont interrompus d'une manière autoritaire par le système. Pour les curieux sachez que durant le context switching, le système d'exploitation place une instruction de saut vers le prochain context switching dans le code qui va être exécuté par le prochain thread. Cette instruction est de type *interruption software*. Si le thread doit s'arrêter avant de rencontrer cette instruction (par exemple parce qu'il est en attente d'une ressource) cette instruction est automatiquement enlevée et le context switching a lieu prématurément.

L'inconvénient majeur du multitâche préemptif est la nécessité de protéger les ressources d'un accès anarchique avec des mécanismes de synchronisation (décrit page 128). Il existe théoriquement un autre modèle de la gestion du multitâche, où la responsabilité de décider quand donner la main incombe au threads eux-mêmes, mais ce modèle est dangereux car les risques de ne jamais rendre la main sont trop grands. Les systèmes d'exploitation Windows n'implémentent plus que le multitâche préemptif.

Les niveaux de priorité d'exécution

Certaines tâches sont plus prioritaires que d'autres. Concrètement elles méritent que le système d'exploitation leur alloue plus de temps processeur. Par exemple, certains pilotes de périphérique, pris en charge par le processeur principal, ne doivent pas être inter-

CS_doNet/05 Page 123 Mercredi, 14. mai 2003 10:32 10

rompus. Une autre catégorie de tâches prioritaires sont les interfaces graphiques utilisateurs. En effet, les utilisateurs n'aiment pas attendre que l'interface se rafraîchisse.

Ceux qui viennent du monde win32, savent bien que le système d'exploitation Windows, sous-jacent au CLR, assigne un numéro de priorité à chaque thread entre 0 et 31. Cependant, il n'est pas dans la philosophie de la gestion des threads sous .NET de travailler directement avec cette valeur, parce que :

- Elle est peu explicite.
- Cette valeur est susceptible de changer au cours du temps.

Niveau de priorité d'un processus

Vous pouvez assigner une priorité à vos processus avec la propriété `PriorityClass` de la classe `System.Diagnostics.Process`. Cette propriété est de type l'énumération `System.Diagnostics.ProcessPriorityClass` qui a les valeurs suivantes :

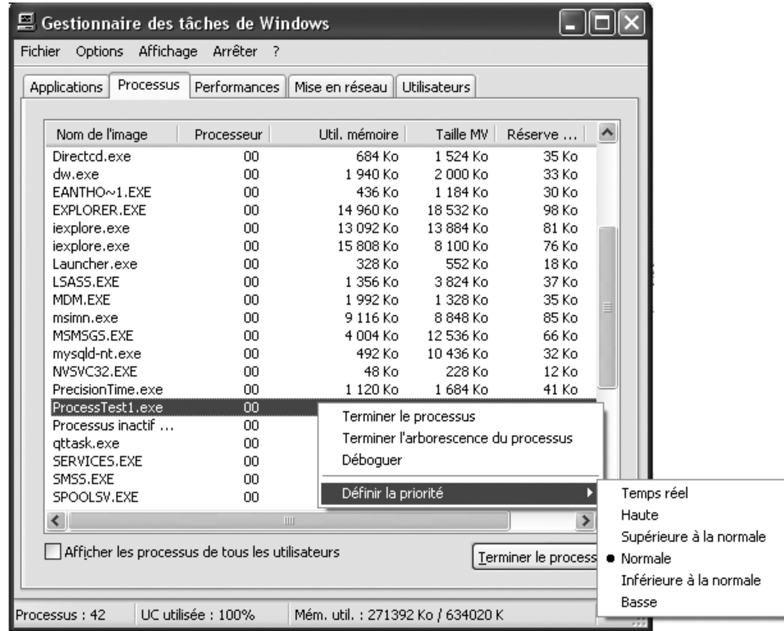
Valeur de `ProcessPriorityClass`	Niveau de priorité correspondant
`Low`	4
`BelowNormal`	6
`Normal`	8
`AboveNormal`	10
`High`	13
`RealTime`	24

Notez que le processus propriétaire de la fenêtre en premier plan voit sa priorité incrémentée d'une unité si la propriété `PriorityBoostEnabled` de la classe `System.Diagnostics.Process` est positionnée à `true`. Cette propriété est par défaut positionnée à `true`. Cette propriété n'est accessible sur une instance de la classe `Process`, que si celle-ci référence un processus sur la même machine.

Vous avez la possibilité de changer la priorité d'un processus en utilisant le *gestionnaire des tâches* comme ceci :

CS_doNet/05 Page 124 Mercredi, 14. mai 2003 10:32 10

Figure 5-2. Changement de la priorité d'un processus



Les systèmes d'exploitation Windows ont un *processus d'inactivité* (*idle* en anglais) qui a la priorité 0. Cette priorité n'est accessible à aucun autre processus. Par définition l'activité des processeurs, notée en pourcentage, est :

100% moins le pourcentage de temps passé dans le thread du processus d'inactivité.

Niveau de priorité d'un thread

Chaque thread peut définir sa propre priorité par rapport à celle de son processus, avec la propriété `Priority` de la classe `System.Threading.Thread`. Cette propriété est de type l'énumération `System.Threading.ThreadPriority` qui présente les valeurs suivantes :

Valeur de `ThreadPriority`	Effet sur la priorité du thread
`Lowest`	-2 unités par rapport à la priorité du processus
`BelowNormal`	–1 unité par rapport à la priorité du processus
`Normal`	même priorité que la priorité du processus
`AboveNormal`	+1 unité par rapport à la priorité du processus
`Highest`	+2 unités par rapport à la priorité du processus

CS_doNet/05 Page 125 Mercredi, 14. mai 2003 10:32 10

Dans la plupart de vos applications, vous n'aurez pas à modifier la priorité de vos processus et threads, qui par défaut, est assignée à `Normal`.

La classe System.Threading.Thread

Chaque thread géré a un objet de la classe `System.Threading.Thread` qui lui est associé automatiquement par le CLR. Vous pouvez utiliser cet objet pour manipuler le thread à partir d'un autre thread ou à partir du thread lui-même. Vous pouvez obtenir cet objet associé au thread courant avec la propriété statique `CurrentThread` de la classe `System.Threading.Thread`:

```
using System.Threading;
...
Thread tcur = Thread.CurrentThread;
```

Une fonctionnalité de la classe `Thread`, bien pratique pour déboguer une application multithread (multitâches), est la possibilité de pouvoir nommer ses threads avec une chaîne de caractères:

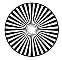
```
tcur.Name = "thread Foo";
```

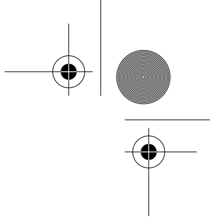
Créer et joindre un thread

Pour créer un nouveau thread dans le processus courant, il suffit de créer un nouvel objet de la classe `Thread`. Le seul constructeur de cette classe prend en argument un objet de type `ThreadStart` qui référence la méthode qui va être exécutée par le thread créé. Une telle classe, qui sert à référencer des méthodes est appelée *délégation*. Les délégations sont décrites page 401. Après sa création le thread ne commence vraiment à s'exécuter qu'à l'appel de la méthode `Start()`.

Exemple 5-3.

```
using System;
using System.Threading;
namespace ThreadTest
{
  class Prog
  {
    static void f1(){Console.WriteLine("hello1");}
    void f2(){Console.WriteLine("hello2");}
    static void Main(string[] args)
    {
      Thread t1 = new Thread(new ThreadStart( f1 ));
      Prog P = new Prog();
      Thread t2 = new Thread(new ThreadStart( P.f2 ));
      t1.Start();
      t2.Start();
      t1.Join();
      t2.Join();
    }
  }
}
```





CS_doNet/05 Page 126 Mercredi, 14. mai 2003 10:32 10

Comme l'illustre cet exemple, la méthode référencée par un délégué de type `ThreadStart` peut être statique ou non.

Dans cet exemple, nous utilisons la méthode `Join()`, qui suspend l'exécution du thread courant jusqu'à ce que le thread sur lequel s'applique cette méthode ait terminé. Cette méthode existe aussi en une version surchargée qui prend en paramètre un entier qui définie le nombre maximal de millisecondes à attendre la fin du thread (i.e un TimeOut). Cette version de `Join()` retourne un booléen positionné à `true` si le thread s'est effectivement terminé.

Suspendre l'activité d'un thread

Vous avez la possibilité de suspendre l'activité d'un thread pour une durée déterminée en utilisant la méthode `Sleep()` de la classe `Thread`. Vous pouvez spécifier la durée au moyen d'un entier qui désigne un nombre de millisecondes ou avec une instance de la structure `System.TimeSpan`. Bien qu'une telle instance puisse spécifier une durée avec la précision du dixième de milliseconde (100 nano-secondes) la granularité temporelle de la méthode `Sleep()` n'est qu'à la milliseconde.

```
// le thread courant est suspendu pour une seconde
Thread.Sleep(1000) ;
```

On peut aussi suspendre l'activité d'un thread en appelant la méthode `Suspend()` de la classe `Thread`, à partir d'un autre thread ou à partir du thread à suspendre. Dans les deux cas le thread se bloque jusqu'à ce qu'un autre thread appelle la méthode `Resume()` de la classe `Thread`. Contrairement à la méthode Sleep(), un appel à `Suspend()` ne suspend pas immédiatement le thread, mais le CLR suspendra ce thread au prochain point protégé rencontré. La notion de point protégé est présentée page 107.

Terminer un thread

Un thread peut se terminer selon trois scénarios :

- Il sort de la méthode sur laquelle il avait commencé (la méthode `Main()` pour le thread principal, la méthode référencée par le délégué `ThreadStart` pour les autres threads).
- Il s'auto-interrompt (il se suicide).
- Il est interrompu par un autre thread.

Le premier cas étant trivial, nous ne nous intéressons qu'aux deux autres cas. Dans ces deux cas, la méthode `Abort()` peut être utilisée (par le thread courant ou par un thread extérieur). Elle provoque l'envoi d'une exception de type `ThreadAbortException`. Cette exception a la particularité d'être relancée automatiquement lorsqu'elle est rattrapée par un gestionnaire d'exception car le thread est dans un état spécial nommé `AbortRequested`. Seul l'appel à la méthode statique `ResetAbort()` (si on dispose de la permission nécessaire) dans le gestionnaire d'exception empêche cette propagation.

CS_doNet/05 Page 127 Mercredi, 14. mai 2003 10:32 10

Exemple 5-4. Suicide du thread principal

```
using System;
using System.Threading;
namespace ThreadTest
{
  class Prog
  {
    static void Main(string[] args)
    {
      Thread t = Thread.CurrentThread;
      try
      {
        t.Abort();
      }
      catch(ThreadAbortException e)
      {
        Thread.ResetAbort();
      }
    }
  }
}
```

Lorsqu'un thread A appelle la méthode `Abort()` sur un autre thread B, il est conseillé que A attende que B soit effectivement terminé en appelant la méthode `Join()` sur B.

Il existe aussi la méthode `Interrupt()` qui permet de terminer un thread lorsqu'il est dans un état d'attente (i.e bloqué sur une des méthodes `Wait()`, `Sleep()` ou `Join()`). Cette méthode a un comportement différent selon que le thread à terminer est dans un état d'attente ou non.

- Si le thread à terminer est dans un état d'attente lorsque `Interrupt()` est appelée par un autre thread, l'exception `ThreadInterruptedException` est lancée.
- Si le thread à terminer n'est pas dans un état d'attente lorsque Interrupt() est appelée, la même exception sera lancée dès que ce thread rentrera dans un état d'attente. Le comportement est le même si le thread à terminer appelle `Interrupt()` sur lui-même.

Notion de threads foreground et background

La classe `Thread` présente la propriété booléenne `IsBackground`. Un *thread foreground* est un thread qui empêche la terminaison du processus tant qu'il n'est pas terminé. A l'opposé un *thread background* est un thread qui est terminé automatiquement par le CLR (par l'appel à la méthode `Abort()`) lorsqu'il n'y a plus de thread foreground dans le processus concerné. `IsBackground` est positionnée à `false` par défaut, ce qui fait que les threads sont foreground par défaut.

On pourrait traduire thread foreground par *thread de premier plan*, et thread background par *thread de fond*.

CS_doNet/05 Page 128 Mercredi, 14. mai 2003 10:32 10

Diagramme d'états d'un thread

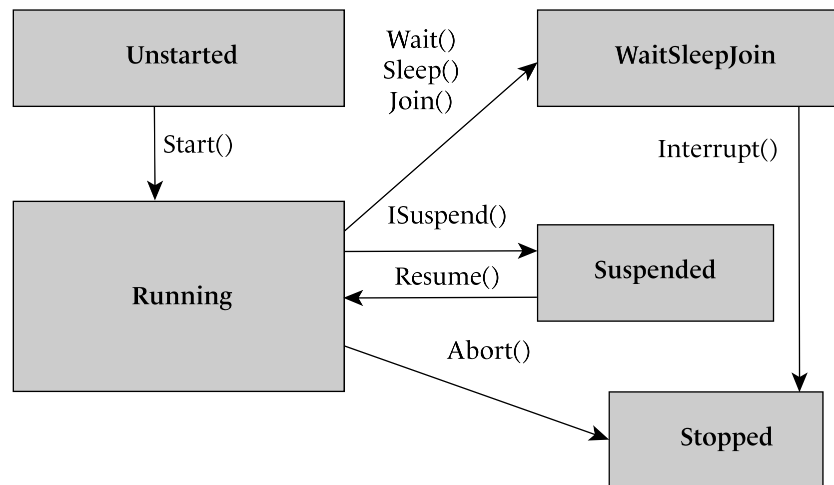
La classe `Thread` a le champ `ThreadState` de type l'énumération `System.Threading.ThreadState`. Les valeurs de cette énumération sont :

```
Aborted              AbortRequested          BackgroundR
Running              Stopped                 StopRequested
Suspended            SuspendRequested        Unstarted
WaitSleepJoin
```

La description de chacun de ces états se trouve dans l'article **ThreadState Enumeration** des **MSDN**. Cette énumération est un *indicateur binaire*, c'est-à-dire que ses instances peuvent prendre plusieurs valeurs à la fois. Par exemple un thread peut être à la fois dans l'état `Running` `AbortRequested` et `Background`. La notion d'indicateur binaire est présentée page 301.

D'après ce qu'on a vu dans la section précédente, on peut définir le diagramme d'état simplifié suivant :

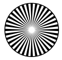
Figure 5-3. Diagramme d'état d'un thread, simplifié

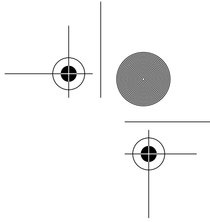


Les objets de synchronisation

Introduction

En informatique, le mot *synchronisation* ne peut être utilisé que dans le cas des applications multithread (mono ou multi-processus). En effet, la particularité de ces applications est d'avoir plusieurs unités d'exécution, d'où possibilité de conflits d'accès aux ressources. Les objets de synchronisation sont des objets partageables entre threads exécutés sur la même machine. Le propre d'un objet de synchronisation est de pouvoir bloquer un des threads utilisateur jusqu'à la réalisation d'une condition par un autre thread.



CS_doNet/05 Page 129 Mercredi, 14. mai 2003 10:32 10


Comme nous allons le voir, il existe de nombreuses classes et mécanismes de synchronisation. Chacun répond à un ou plusieurs besoins spécifiques et il est nécessaire d'avoir assimilé tout ce chapitre avant de concevoir une application professionnelle multithreads utilisant la synchronisation. Nous nous sommes efforcé de souligner les différences, surtout les plus subtiles, qui existent entre les différents mécanismes. Quand vous aurez compris les différences, vous serez capable d'utiliser ces mécanismes.

Synchroniser correctement un programme est une des tâches du développement logiciel les plus subtiles. Le sujet remplit de nombreux ouvrages. Avant de vous plonger dans des spécifications compliquées, soyez certains que l'utilisation de la synchronisation est incontournable. Souvent l'utilisation de quelques règles simples suffit à éviter d'avoir à gérer la synchronisation. Parmi ces règles, citons la règle d'affinité entre threads et ressources. Cette règle dit que pour éviter d'avoir à gérer la synchronisation des accès à une ressource, il suffit de faire en sorte qu'elle soit toujours accédée par le même thread.

Notion de deadlocks et de race conditions

Avant d'aborder les mécanismes de synchronisation, il est nécessaire d'avoir une idée précise des notions de *race conditions* (*situations de compétition* en français) et de *deadlocks* (*interblocages* en français).

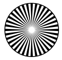
Race conditions

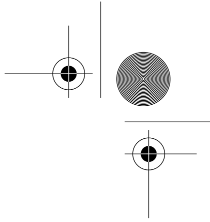
Il s'agit d'une situation où des actions effectuées par des unités d'exécution différentes s'enchaînent dans un ordre illogique, entraînant des états non prévus.

Par exemple un thread T modifie une ressource R, rend les droits d'accès d'écriture à R, reprend les droits d'accès en lecture sur R et utilise R comme si son état était celui dans lequel il l'avait laissé. Pendant l'intervalle de temps entre la libération des droits d'accès en écriture et l'acquisition des droits d'accès en lecture, il se peut qu'un autre thread ait modifié l'état de R.

Un autre exemple classique de situation de compétition est le modèle producteur consommateur. Le producteur utilise souvent le même espace physique pour stocker les informations produites. En général on n'oublie pas de protéger cet espace physique des accès concurrents entre producteurs et consommateurs. On oublie plus souvent que le producteur doit s'assurer qu'un consommateur a effectivement lu une ancienne information avant de produire une nouvelle information. Si l'on ne prend pas cette précaution, on s'expose au risque de produire des informations qui ne seront jamais consommées.

Les conséquences de situations de compétition mal gérées peuvent être des failles dans un système de sécurité. Une autre application peut forcer un enchaînement d'actions non prévues par les développeurs. Typiquement il faut absolument protéger l'accès en écriture à un booléen qui confirme ou infirme une authentification. Sinon il se peut que son état soit modifié entre l'instant ou ce booléen est positionné par le mécanisme d'authentification et l'instant ou ce booléen est lu pour protéger des accès à des ressources. De célèbres cas de failles de sécurité dues à une mauvaise gestion des situations de compétition ont existé. Une de celles-ci concernait notamment le noyau *Unix*.





CS_doNet/05 Page 130 Mercredi, 14. mai 2003 10:32 10

Deadlocks

Il s'agit d'une situation de blocage à cause de deux ou plusieurs unités d'exécution qui s'attendent mutuellement.

Par exemple :

Un thread T1 acquiert les droits d'accès sur la ressource R1.
Un thread T2 acquiert les droits d'accès sur la ressource R2.
T1 demande les droits d'accès sur R2 et attend, car c'est T2 qui les possède.
T2 demande les droits d'accès sur R1 et attend, car c'est T1 qui les possède.

T1 et T2 attendront donc indéfiniment, la situation est bloquée ! Il existe trois techniques différentes pour éviter ce problème qui est plus subtil que la plupart des bugs que l'on rencontre.

- N'autoriser aucun thread à avoir des droits d'accès sur plusieurs ressources simultanément.
- Définir une relation d'ordre dans l'acquisition des droits d'accès aux ressources. C'est-à-dire qu'un thread ne peut acquérir les droits d'accès sur R2 s'il n'a pas déjà acquis les droits d'accès sur R1. Naturellement la libération des droits d'accès se fait dans l'ordre inverse de l'acquisition.
- Systématiquement définir un temps maximum d'attente (*timeout*) pour toutes les demandes d'accès aux ressources et traiter les cas d'échec. Pratiquement tous les mécanismes de synchronisation .NET offrent cette possibilité.

Les deux premières techniques sont plus efficaces mais aussi plus difficile à implémenter. En effet, elles nécessitent chacune une contrainte très forte et difficile à maintenir durant l'évolution de l'application. En revanche les situations d'échecs sont inexistantes.

Les gros projets utilisent systématiquement la troisième technique. En effet, si le projet est gros, le nombre de ressources est en général très grand. Dans ces projets, les conflits d'accès simultanés à une ressource sont donc des situations marginales. La conséquence est que les situations d'échec sont, elles-aussi, marginales.

Les champs volatiles

Un champ d'un type peut être accédé par plusieurs threads. Supposons que ces accès, en lecture ou en écriture, ne soient pas synchronisés. Dans ce cas, les nombreux mécanismes internes de la gestion du code à l'exécution, font qu'il n'y a pas de garantie que chaque accès en lecture au champ charge la valeur la plus récente. Un champ déclaré volatile vous donne cette garantie. En langage C#, un champ est déclaré volatil si le mot-clé `volatile` est écrit devant sa déclaration.

Tous les champs ne peuvent pas être volatiles. Il y a une restriction sur le type du champ. Pour qu'un champ puisse être volatile, il faut que son type soit dans cette liste :

- Un type référence.
- Un pointeur (dans une zone de code non protégée).
- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, `bool` (`double`, `long` et `ulong`, à la condition de travailler avec une machine 64 bits).

- Une énumération dont le type sous-jacent est parmi : `byte`, `sbyte`, `short`, `ushort`, `int`, `uint` (`double`, `long` et `ulong`, à la condition de travailler avec une machine 64 bits).

Comme vous l'aurez remarquez, seuls les types dont la valeur ou la référence fait au plus le nombre d'octets d'un entier natif (quatre ou huit), peuvent être volatiles. Cela implique que les opérations concurrentes sur une valeur de plus de ce nombre d'octets (une grosse structure par exemple) doit utiliser les mécanismes de synchronisation présentés ci-après.

La classe System.Threading.Interlocked

L'expérience a montré que les ressources à protéger dans un contexte multi threads sont très souvent des variables entières. Les opérations les plus courantes réalisées par les threads sur ces variables entières partagées, sont l'incrémentation et la décrémentation d'une unité. Le Framework .NET prévoit donc un mécanisme spécial avec la classe `System.Threading.Interlocked` pour ces opérations très spécifiques, mais aussi très courantes. Cette classe à deux méthodes statiques `Increment()` et `Decrement()`, qui incrémente ou décrémente une variable entière de type `int` ou `long` passée par référence. On dit que l'utilisation de la classe `Interlocked` rend ces opérations *atomiques* (c'est-à-dire indivisibles, comme ce que l'on pensait il y a quelques décennies pour les atomes).

Le programme suivant présente l'accès concurrent de deux threads à la variable entière `Compteur`. Un thread l'incrémente cinq fois tandis que l'autre la décrémente cinq fois.

Exemple 5-5.

```
using System;
using System.Threading;
namespace InterlockedTest
{
  class Prog
  {
    static long Compteur =1;
    static void Main(string[] args)
    {
      Thread t1 = new Thread(new ThreadStart( f1 ));
      Thread t2 = new Thread(new ThreadStart( f2 ));
      t1.Start();t2.Start();
      t1.Join(); t2.Join();
    }
    static void f1()
    {
      for(int i =0;i<5;i++)
      {
        Interlocked.Increment( ref Compteur );
        Console.WriteLine("Compteur++ {0}",Compteur);
        Thread.Sleep(10);
      }
    }
```

CS_doNet/05 Page 132 Mercredi, 14. mai 2003 10:32 10


```
    static void f2()
    {
      for(int i =0;i<5;i++)
      {
        Interlocked.Decrement( ref Compteur );
        Console.WriteLine("Compteur-- {0}",Compteur);
        Thread.Sleep(10);
      }
    }
  }
}
```

Ce programme affiche ceci (d'une manière non déterministe, c'est-à-dire que l'affichage pourrait varier d'une exécution à une autre) :

```
Compteur++ 2
Compteur-- 1
Compteur++ 2
Compteur-- 1
Compteur++ 2
Compteur-- 1
Compteur++ 2
Compteur-- 1
Compteur-- 0
Compteur++ 1
```

Si on n'endormait pas les threads 10 millièmes de seconde à chaque modification, les threads auraient le temps de réaliser leurs tâches en un quantum et il n'y aurait pas l'entrelacement des exécutions, donc pas d'accès concurrent.

Autre possibilité d'utilisation de la classe Interlocked

La classe `Interlocked` permet de rendre atomique une autre opération usuelle qui est la recopie de l'état d'un objet source vers un objet destination au moyen de la méthode statique surchargée `Exchange()`. Elle permet aussi de rendre atomique l'opération de comparaison des états de deux objets, et dans le cas d'égalité, la recopie de cet état vers un troisième objet au moyen de la méthode statique surchargée `CompareExchange()`. (Voir les MSDN pour plus de détails.)

La classe System.Threading.Monitor et le mot-clé lock

Le fait de rendre des opérations simples atomiques (des opérations comme l'incrémentation, la décrémentation ou la recopie d'un état), est indéniablement important mais est loin de couvrir tous les cas où la synchronisation est nécessaire. La classe `System.Threading.Monitor` permet de rendre n'importe quelle portion de code exécutable par un seul thread à la fois. On appelle une telle portion de code une *section critique*.

Les méthodes Enter() et Exit()

La classe `Monitor` présente les méthodes statiques `Enter(object)` et `Exit(object)`. Ces méthodes prennent un objet en paramètre. Cet objet constitue un moyen simple

CS_doNet/05 Page 133 Mercredi, 14. mai 2003 10:32 10

d'identifier de manière unique la ressource à protéger d'un accès concurrent. Lorsqu'un thread appelle la méthode `Enter()`, il attend d'avoir le droit exclusif de posséder l'objet référencé (il n'attend que si un thread a déjà ce droit). Une fois ce droit acquis et consommé, le thread libère ce droit en appelant `Exit()` sur ce même objet.

En général on veut protéger soit un objet particulier soit une classe particulière, des accès concurrents. Le meilleur moyen est :

- d'utiliser les méthodes `Enter()` et `Exit()` avec la référence `this` dans les méthodes partagées de l'objet à protéger,
- d'utiliser les méthodes `Enter()` et `Exit()` avec la référence `typeof` ('le type de la classe courante') dans les méthodes statiques de la classe à protéger.

Un thread peut appeler `Enter()` plusieurs fois sur le même objet à la condition qu'il appelle `Exit()` autant de fois sur le même objet pour se libérer des droits exclusifs.
Un thread peut posséder des droits exclusifs sur plusieurs objets à la fois, mais cela peut mener au problème connu sous le nom de dead lock, présenté un peu plus haut.
Il ne faut jamais appeler les méthodes `Enter()` et `Exit()` sur un objet de type valeur, comme un entier !
Il faut toujours appeler `Exit()` dans un bloc `finally` afin d'être certain de libérer les droits d'accès exclusifs quoi qu'il arrive.

Si dans l'exemple de la section page 130, un thread doit élever la variable `Compteur` au carré tandis que l'autre thread doit la multiplier par deux, il faudrait remplacer l'utilisation de la classe `Interlocked` par l'utilisation de la classe `Monitor`. Le code de `f1()` et `f2()` serait alors :

Exemple 5-6.

```
...
static void f1()
{
  for(int i =0;i<5;i++)
  {
    try
    {
      Monitor.Enter( typeof(Prog) );
      Compteur*=Compteur;
    }
    finally
    {
      Monitor.Exit( typeof(Prog) );
    }
    Console.WriteLine("Compteur^2 {0}",Compteur);
    Thread.Sleep(10);
  }
```

CS_doNet/05 Page 134 Mercredi, 14. mai 2003 10:32 10


```
}
static void f2()
{
  for(int i =0;i<5;i++)
  {
    try
    {
      Monitor.Enter( typeof(Prog) );
      Compteur*=2;
    }
    finally
    {
      Monitor.Exit( typeof(Prog) );
    }
    Console.WriteLine("Compteur*2 {0}",Compteur);
    Thread.Sleep(10);
  }
}
...
```

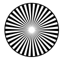
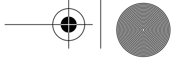
Il est tentant d'écrire `Compteur` à la place de `typeof(Prog)` mais Compteur est un membre statique de type valeur. De plus les opérations 'élévation au carré' et 'multiplication par deux' n'étant pas commutatives, la valeur finale de `Compteur` est ici non déterminée.

Le mot clé lock de C#

Le langage C# présente le mot-clé lock qui remplace élégamment l'utilisation des méthode `Enter()` et `Exit()`. La méthode `f1()` pourrait donc s'écrire :

Exemple 5-7.

```
...
static void f1()
{
  for(int i =0;i<5;i++)
  {
    lock( typeof(Prog) )
    {
      Compteur*=Compteur;
    }
    Console.WriteLine("Compteur^2 {0}",Compteur);
    Thread.Sleep(10);
  }
}
...
```



CS_doNet/05 Page 135 Mercredi, 14. mai 2003 10:32 10

A l'instar des blocs `for` et `if`, les blocs définis par le mot-clé lock ne sont pas tenus d'avoir des accolades s'ils ne contiennent qu'une instruction. On aurait donc pu écrire :

```
...
lock( typeof(Prog) )
  Compteur*=Compteur;
...
```

Autres méthodes de la classe Monitor

La classe `Monitor` présente d'autres méthodes statiques qui lui confèrent des fonctionnalités inaccessibles par l'utilisation des seules méthodes `Enter()` et `Exit()`.

```
bool TryEnter(object [,int] )
```

Cette méthode est similaire à `Enter()` mais elle n'est pas bloquante. Si les droit d'accès exclusifs sont déjà possédés par un autre thread, cette méthode retourne immédiatement et sa valeur de retour est `false`. On peut aussi rendre un appel à `TryEnter()` bloquant pour une durée limitée spécifiée en millisecondes. Puisque l'issue de cette méthode est incertaine, et que dans le cas où l'on acquerrait les droits d'accès exclusifs il faudrait les libérer dans un bloc `finally`, il est conseillé de sortir immédiatement de la méthode courante dans le cas où l'appel `TryEnter()` échouerait :

```
...
try
{
  if( ! Monitor.TryEnter(x) )
    return;
  // ...
}
finaly
{
  Monitor.Exit(x);
}
...

Wait(object [,int]) ; Pulse(object); PulseAll(object)
```

Les trois méthodes `Wait()` `Pulse()` et `PulseAll()` doivent être utilisées ensembles et ne peuvent être correctement comprises sans un petit scénario. L'idée est qu'un thread ayant les droits d'accès exclusifs à un objet décide d'attendre (en appelant `Wait()`) que l'état de l'objet change. Pour cela, ce thread doit accepter de perdre momentanément les droit d'accès exclusifs à l'objet afin de permettre à un autre thread de changer l'état de l'objet. Ce dernier doit signaler le changement avec la méthode `Pulse()`. Voici un petit scénario expliquant ceci dans les détails :

- Le thread T1 possédant l'accès exclusif à l'objet OBJ, appelle la méthode `Wait(OBJ)` afin de s'enregistrer dans une liste d'attente passive de OBJ.
- Par cet appel T1 perd l'accès exclusif à OBJ. Ainsi un autre thread T2 prend l'accès exclusif à OBJ en appelant la méthode `Enter(OBJ)`.

CS_doNet/05 Page 136 Mercredi, 14. mai 2003 10:32 10

- T2 modifie éventuellement l'état de OBJ puis appelle `Pulse(OBJ)` pour signaler cette modification. Cet appel provoque le passage du premier thread de la liste d'attente passive de OBJ (en l'occurrence T1) en haut de la liste d'attente active de OBJ. Le premier thread de la liste active de OBJ a la garantie qu'il sera le prochain à avoir les droits d'accès exclusifs à OBJ dès qu'ils seront libérés. Il pourra ainsi sortir de son attente dans la méthode `Wait(OBJ)`.
- Dans notre scénario T2 libère les droits d'accès exclusif sur OBJ en appelant `Exit(OBJ)` et T1 les récupère et sort de la méthode `Wait(OBJ)`.
- La méthode `PulseAll()` fait en sorte que les threads de la liste d'attente passive, passent tous dans la liste d'attente active. L'important est que les threads soient débloqués dans le même ordre qu'ils ont appelés `Wait()`.

Si `Wait(OBJ)` est appelée par un thread ayant appelé plusieurs fois `Enter(OBJ)`, ce thread devra appeler `Exit(OBJ)` le même nombre de fois pour libérer les droits d'accès à OBJ. Même dans ce cas, un seul appel à `Pulse(OBJ)` par un autre thread suffit à débloquer le premier thread.

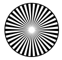
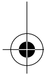
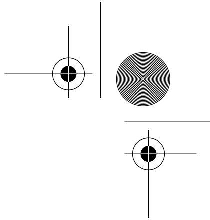
Le programme suivant résume le scénario exposé, à la différence qu'il est obligé de tenir compte du fait que T2 peut potentiellement acquérir les droits d'accès avant T1. Pour pallier cela on se sert d'un booléen qui signale si T1 a déjà eu les droits d'accès. T1 prend les droits d'accès, change le booléen, et 'Pulse T2' qui est peut être en train d'attendre que T1 ait acquis les droits d'accès.

Exemple 5-8.

```
using System;
using System.Threading;

public class ClasseDeOBJ
{
  public int MyState;
  public override string ToString() { return MyState.ToString();}
}

public class Prog
{
  static ClasseDeOBJ OBJ;
  static bool bT1AAcquisLesDroitsExcusifsSurOBJ = false;
  public static void Main(string[] args)
  {
  OBJ = new ClasseDeOBJ();
  Thread T1 = new Thread(new ThreadStart(T1Proc));
  Thread T2 = new Thread(new ThreadStart(T2Proc));
  T2.Start(); T1.Start();
  T1.Join(); T2.Join();
  }
```

CS_doNet/05 Page 137 Mercredi, 14. mai 2003 10:32 10


```
static void T1Proc()
{
  Console.WriteLine("T1: Hello!");
  try
  {
  Monitor.Enter(OBJ);
  Console.WriteLine("T1: J'ai acquis les drois d'accès sur OBJ");
  bT1AAcquisLesDroitsExcusifsSurOBJ = true;
  Monitor.Pulse(OBJ);// signale à T2 que j'ai obtenu les drois sur OBJ
  Console.WriteLine("T1: OBJ vaut {0} j'attends que son état change",OBJ);
  Monitor.Wait(OBJ);// attend que T2 ait changé l'état de OBJ
                  Console.WriteLine("T1: OBJ vaut {0}",OBJ);
  }
  finally
  {
  Monitor.Exit(OBJ);
  }
  Console.WriteLine("T1: Bye!");
}

static void T2Proc()
{
  Console.WriteLine("T2: Hello!");
  try
  {
  Monitor.Enter(OBJ);
  Console.WriteLine("T2: J'ai acquis les drois d'accès exclusifs sur OBJ");
  if( !bT1AAcquisLesDroitsExcusifsSurOBJ )
  {
   Console.WriteLine("T2: T1 n'a pas encore eu l'accès à OBJ, j'attend...");
   Monitor.Wait(OBJ);// attend que T1 ait obtenu les drois sur OBJ
  }
  OBJ.MyState = 1;
  Console.WriteLine("T2: J'ai modifié OBJ et je le signale avec Pulse().");
  Monitor.Pulse(OBJ);// signale à T1 que OBJ à changé d'état
  }
  finally
  {
  Console.WriteLine("T2: Je vais libérer les drois d'accès sur OBJ.");
  Monitor.Exit(OBJ);
  }
  Console.WriteLine("T2: Bye!");
}
}
```

CS_doNet/05 Page 138 Mercredi, 14. mai 2003 10:32 10

Si T1 obtient les droits d'accès exclusifs sur OBJ avant T2 le programme affiche :

```
T1: Hello!
T1: J'ai acquis les drois d'accès  sur OBJ
T1: OBJ vaut 0 j'attends que son état change...
T2: Hello!
T2: J'ai acquis les drois d'accès exclusifs sur OBJ
T2: J'ai modifié OBJ et je le signale avec Pulse().
T2: Je vais libérer les drois d'accès sur OBJ.
T1: OBJ vaut 1
T1: Bye!
T2: Bye!
```

Si T2 obtient les droits d'accès exclusifs sur OBJ avant T1 le programme affiche :

```
T2: Hello!
T2: J'ai acquis les drois d'accès exclusifs sur OBJ
T2: T1 n'a pas encore eu l'accès à OBJ, j'attend...
T1: Hello!
T1: J'ai acquis les drois d'accès  sur OBJ
T1: OBJ vaut 0 j'attends que son état change...
T2: J'ai modifié OBJ et je le signale avec Pulse().
T2: Je vais libérer les drois d'accès sur OBJ.
T2: Bye!
T1: OBJ vaut 1
T1: Bye!
```

Mutex et événements

La classe de base abstraite `System.Threading.WaitHandle` admet trois classes dérivées, dont l'utilisation est bien connue de ceux qui ont déjà utilisé la synchronisation sous win32 :

- La classe `Mutex` (le mot mutex est la concaténation de mutuelle exclusion. En français on parle parfois de *mutant*)).
- La classe `AutoResetEvent` qui définit un événement à repositionnement automatique.
- La classe `ManualResetEvent` qui définit un événement à repositionnement manuelle.

La classe WaitHandle et ses classes dérivées, ont la particularité d'implémenter la méthode non statique `WaitOne()` et les méthodes statiques `WaitAll()` et `WaitAny()`. Elles permettent respectivement d'attendre qu'un objet soit signalé, que tous les objets dans un tableau soient signalés, qu'au moins un objet dans un tableau soit signalé. Contrairement à la classe `Monitor` et `Interlocked`, ces classes doivent être instanciées pour être utilisées. Il faut donc raisonner ici en terme d'objets de synchronisation et non d'objets synchronisés. Ceci implique que les objets passés en paramètre des méthodes statiques `WaitAll()` et `WaitAny()` sont soit des mutex, soit des événements.

Il est important de noter une autre grosse distinction entre l'utilisation des classes dérivées de `WaitHandle` et l'utilisation de la classe `Monitor`. L'utilisation de la classe `Monitor` est totalement gérée par le CLR et se cantonne à un seul processus. L'utilisation

CS_doNet/05 Page 139 Mercredi, 14. mai 2003 10:32 10

des classes dérivées de `WaitHandle` fait appel (dans la version actuelle 1.0 et 1.1 de .NET) à des objets win32 non gérés. De plus, un même mutex peut être connu de plusieurs processus de la même machine. L'utilisation de tels objets est donc plus coûteuse.

Les Mutex

En terme de fonctionnalités les mutex sont proches de l'utilisation de la classe `Monitor` à ces différences près :

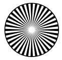
- En nommant un mutex, on peut l'utiliser dans plusieurs processus d'une même machine.
- L'utilisation de `Monitor` ne permet pas de se mettre en attente sur plusieurs objets.
- Les mutex n'ont pas la fonctionnalité des méthodes `Wait()`, `Pulse()` et `PulseAll()` de la classe `Monitor`.

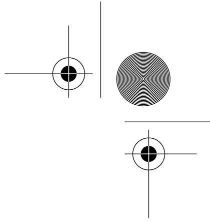
Le programme suivant montre l'utilisation d'un *mutex nommé* pour protéger l'accès à une ressource partagée par plusieurs processus sur la même machine. La ressource partagée est un fichier où chaque programme écrit 10 lignes.

Exemple 5-9.

```
using System;
using System.Threading;
using System.IO;

class Prog
{
  static void Main(string[] args)
  {
    // Le mutex est nommé 'MutexTest'
    Mutex MutexFile = new Mutex(false,"MutexTest");
    for(int i = 0; i<10 ;i++)
    {
      MutexFile.WaitOne();
      // Ouvre le fichier, écrit Hello i, ferme le fichier
      FileInfo FI = new FileInfo("tmp.txt");
      StreamWriter SW = FI.AppendText();
      SW.WriteLine("Hello {0}",i);
      SW.Flush();
      SW.Close();
      // attend 1 seconde
      // pour rendre évidente l'action du mutex
      Console.WriteLine("Hello {0}",i);
      Thread.Sleep(1000);
      MutexFile.ReleaseMutex();
    }
    MutexFile.Close();
  }
}
```

CS_doNet/05 Page 140 Mercredi, 14. mai 2003 10:32 10





Notez bien l'utilisation de la méthode `WaitOne()` qui bloque le thread courant jusqu'à l'obtention du mutex et l'utilisation de la méthode `ReleaseMutex()` qui libère le mutex.

Notez que dans ce programme, `new Mutex` ne signifie pas forcément la création du mutex mais la création d'une référence vers le mutex nommé `"MutexTest"`. Le mutex est effectivement créé par le système d'exploitation seulement s'il n'existe pas déjà. De même la méthode `Close()` ne détruit pas forcément le mutex si ce dernier est encore référencé par d'autres processus.

Pour ceux qui avaient l'habitude d'utiliser la technique du mutex nommé en win32, pour éviter de lancer deux instances du même programme sur la même machine, sachez qu'il existe une meilleure façon de procéder sous .NET, décrite dans la section page 120.

Les événements

Contrairement aux mécanismes de synchronisation vus jusqu'ici, les événements ne définissent pas explicitement de notion d'appartenance d'une ressource à un thread à un instant donné. Les événements servent à passer une notification d'un thread à l'autre, cette notification étant 'un événement s'est passé'. L'événement concerné est associé à une instance d'une des deux classes d'événement, `System.Threading.AutoResetEvent` et `System.Threading.ManualResetEvent`.

Concrètement un thread attend qu'un événement soit signalé en appelant la fonction bloquante `WaitOne()` sur l'objet événement associé. Puis un thread signale l'événement en appelant la méthode `Set()` sur l'objet événement associé et permet ainsi au premier thread de reprendre son exécution.

La différence entre les événements `AutoResetEvent` (*événement à repositionnement automatique*) et `ManualResetEvent` (*événement à repositionnement manuel*) est que l'on a besoin d'appeler la méthode `Reset()` pour repositionner l'événement en position non active, sur un événement de type 'à repositionnement manuel' après l'avoir signalé.

La différence entre le repositionnement manuel et automatique est plus importante que l'on pourrait croire. Si plusieurs threads attendent sur un même événement à repositionnement automatique, il faut signaler l'événement une fois pour chaque thread. Dans le cas d'un événement à repositionnement manuel il suffit de signaler une fois l'événement pour débloquer tous les threads.

Le programme suivant crée deux threads, T0 et T1, qui incrémentent chacun leur propre compteur à des vitesses différentes. T0 signale l'événement EV[0] lorsqu'il est arrivé à 5 et T1 signale l'événement EV[1] lorsqu'il est arrivé à 8. Le thread principal attend que les deux événements soient signalés pour afficher un message.

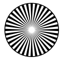
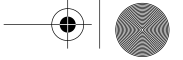
CS_doNet/05 Page 141 Mercredi, 14. mai 2003 10:32 10

Exemple 5-10.

```
using System;
using System.Threading;

class Prog
{
  static AutoResetEvent[] EV;
  static void Main(string[] args)
  {
    EV = new AutoResetEvent[2];
    // position initiale de l'événement: false
    EV[0] = new AutoResetEvent(false);
    EV[1] = new AutoResetEvent(false);
    Thread T0 = new Thread(new ThreadStart(ThreadProc0));
    Thread T1 = new Thread(new ThreadStart(ThreadProc1));
    T0.Start(); T1.Start();
    AutoResetEvent.WaitAll(EV);
    Console.WriteLine("MainThread: Thread0 est arrivé à 5"+
      "et Thread1 est arrivé à 8");
    T0.Join();
    T1.Join();
  }
  static void ThreadProc0()
  {
    for(int i=0;i<10;i++)
    {
      Console.WriteLine("Thread0: {0}",i);
      if( i == 5 )EV[0].Set();
      Thread.Sleep(100);
    }
  }
  static void ThreadProc1()
  {
    for(int i=0;i<10;i++)
    {
      Console.WriteLine("Thread1: {0}",i);
      if( i == 8 )EV[1].Set();
      Thread.Sleep(60);
    }
  }
}
```

Ce programme affiche :

```
Thread0: 0
Thread1: 0
Thread1: 1
Thread0: 1
Thread1: 2
Thread1: 3
Thread0: 2
```



CS_doNet/05 Page 142 Mercredi, 14. mai 2003 10:32 10


```
Thread1: 4
Thread0: 3
Thread1: 5
Thread1: 6
Thread0: 4
Thread1: 7
Thread1: 8
Thread0: 5
MainThread: Thread0 est arrivé à 5 et Thread1 est arrivé à 8
Thread1: 9
Thread0: 6
Thread0: 7
Thread0: 8
Thread0: 9
```

La classe System.Threading.ReaderWriterLock

La classe `System.Threading.ReaderWriterLock` implémente le mécanisme de synchronisation 'accès lecture multiple/accès écriture unique'. Contrairement au modèle de synchronisation 'accès exclusif' offert par la classe `Monitor` ou `Mutex`, ce mécanisme tient compte du fait qu'un thread demande les droits d'accès en lecture ou en écriture. Un accès en lecture peut être obtenu lorsqu'il n'y a pas d'accès en écriture courant. Un accès en écriture peut être obtenu lorsqu'il n'y a aucun accès courant, ni en lecture ni en écriture. De plus lorsqu'un accès en écriture a été demandé mais pas encore obtenu, les nouvelles demandes d'accès en lecture sont mises en attente.

Lorsque ce modèle de synchronisation peut être appliqué, il faut toujours le privilégier par rapport au modèle proposé par les classes `Monitor` ou `Mutex`. En effet, le modèle 'accès exclusif' ne permet en aucun cas des accès simultanés et est donc moins performant. De plus, empiriquement, on se rend compte que la plupart des applications accèdent beaucoup plus souvent aux données en lecture qu'en écriture.

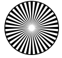
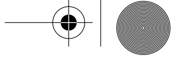
A l'instar des mutex et des événements et au contraire de la classe `Monitor`, la classe `ReaderWriterLock` doit être instanciée pour être utilisée. Il faut donc ici aussi raisonner en terme d'objets de synchronisation et non d'objets synchronisés.

Voici un exemple de code qui montre l'utilisation de cette classe, mais qui n'en exploite pas toutes les possibilités. En effet les méthodes `DowngradeFromWriterLock()` et `UpgradeToWriterLock()` (non détaillées ici, voir les MSDN) permettent de demander un changement de droit d'accès sans avoir à libérer son droit d'accès courant.

CS_doNet/05 Page 143 Mercredi, 14. mai 2003 10:32 10

Exemple 5-11.

```
using System;
using System.Threading;

class Prog
{
  static int LaRessource = 0;
  static ReaderWriterLock RWL;
  static void Main(string[] args)
  {
    RWL = new ReaderWriterLock();
    Thread TR0 = new Thread(new ThreadStart(ThreadReader));
    Thread TR1 = new Thread(new ThreadStart(ThreadReader));
    Thread TW  = new Thread(new ThreadStart(ThreadWriter));
    TR0.Start(); TR1.Start(); TW.Start();
    TR1.Join();  TR1.Join();  TW.Join();
  }
  static void ThreadReader()
  {
    for(int i=0;i<5;i++)
    {
      RWL.AcquireReaderLock(1000);
      Console.WriteLine(
        "Début lecture  LaRessource={0}",LaRessource);
      Thread.Sleep(5);
      Console.WriteLine(
        "Fin   lecture  LaRessource={0}",LaRessource);
      RWL.ReleaseReaderLock();
    }
  }
  static void ThreadWriter()
  {
    for(int i=0;i<5;i++)
    {
      RWL.AcquireWriterLock(1000);
      Console.WriteLine(
        "Début écriture LaRessource={0}",LaRessource);
      Thread.Sleep(100);
      LaRessource++;
      Console.WriteLine(
        "Fin   écriture LaRessource={0}",LaRessource);
      RWL.ReleaseWriterLock();
    }
  }
}
```

CS_doNet/05 Page 144 Mercredi, 14. mai 2003 10:32 10

Ce programme affiche :

```
Début lecture  LaRessource=0
Début lecture  LaRessource=0
Fin   lecture  LaRessource=0
Fin   lecture  LaRessource=0
Début écriture LaRessource=0
Fin   écriture LaRessource=1
Début lecture  LaRessource=1
Début lecture  LaRessource=1
Fin   lecture  LaRessource=1
Fin   lecture  LaRessource=1
Début écriture LaRessource=1
Fin   écriture LaRessource=2
Début lecture  LaRessource=2
Début lecture  LaRessource=2
Fin   lecture  LaRessource=2
Fin   lecture  LaRessource=2
Début écriture LaRessource=2
Fin   écriture LaRessource=3
Début lecture  LaRessource=3
Début lecture  LaRessource=3
Fin   lecture  LaRessource=3
Fin   lecture  LaRessource=3
Début écriture LaRessource=3
Fin   écriture LaRessource=4
Début lecture  LaRessource=4
Début lecture  LaRessource=4
Fin   lecture  LaRessource=4
Fin   lecture  LaRessource=4
Début écriture LaRessource=4
Fin   écriture LaRessource=5
```

Accès synchronisés aux éléments d'une collection

Les classes suivantes, représentant un type de collection, présentent la fonctionnalité d'avoir éventuellement un accès synchronisé à leurs éléments :

```
Sytem.Collections.ArrayList
Sytem.Collections.Queue
Sytem.Collections.Stack
Sytem.Collections.SortedList
Sytem.Collections.Hashtable
```

Les classes suivantes ne présentent pas d'accès synchronisé à leurs éléments :

```
System.Array
System.Collection.BitArray
```

Notez que la présentation des collections fait l'objet du chapitre 14, page 431.

CS_doNet/05 Page 145 Mercredi, 14. mai 2003 10:32 10

Cette fonctionnalité est possible grâce à la méthode Synchronized() qui retourne un 'wrapper' d'accès synchronisé de la collection. Ce 'wrapper' est de la même classe que la collection. Aussi, vous pouvez différencier une collection synchronisée d'une collection non synchronisée grâce à la propriété, accessible en lecture seule, bool IsSynchronized() de l'interface System.Collections.ICollection. Cette interface est implémentée par toutes les classes de collection. Voici un exemple pour illustrer tout ceci :

Exemple 5-12.

```
using System.Collections;

public class Prog
{
  public static void Main()
  {
    ArrayList  Liste1 = new ArrayList();
    Liste1.Add( 1 );
    Liste1.Add( 2 );
    ArrayList  Liste2 = ArrayList.Synchronized(Liste1);
    bool b1 = Liste1.IsSynchronized;
    bool b2 = Liste2.IsSynchronized;
    // b1 vaut false et b2 vaut true,
    // Liste2 est un wrapper synchronisé de Liste1
  }
}
```

En terminologie *design pattern*, on dit que l'on 'décore' la collection. Cela signifie que l'on ajoute des fonctionnalités à un objet (en l'occurrence l'accès synchronisé à une collection) en interceptant les appels à l'aide d'objets présentant les même méthodes.

Autre façon pour synchroniser une collection

Le parcours des éléments d'un tableau ne peut pas être synchronisé par l'utilisation d'un wrapper. En effet, le wrapper synchronise les accès et non pas les ensembles d'accès. Par exemple supposons que le thread T1 parcourt une collection et que le thread T2 modifie la collection en même temps. Il y a de fortes chances pour que le parcours de T1 soit perturbé, ce qui mène en général au lancement d'une exception. Pour se prémunir de cela, vous pouvez synchroniser un ensemble d'accès à une collection en utilisant la propriété, accessible en lecture seule, object SyncRoot() de l'interface ICollection. Cette interface est implémentée par toutes les classes de collection. La méthode SyncRoot() retourne un objet utilisable par un objet de synchronisation Monitor (c'est-à-dire le mot-clé du langage C# lock). On peut ainsi synchroniser avec cette technique les accès aux collections qui ne présentent pas la méthode Synchronized(). L'exemple suivant synchronise l'énumération des éléments d'un tableau d'entier :

CS_doNet/05 Page 146 Mercredi, 14. mai 2003 10:32 10

Exemple 5-13.

```
using System;

public class Prog
{
  public static void Main()
  {
    int []  Tab = {1,2,3};
    lock( Tab.SyncRoot )
    {
      foreach( int i in Tab )
      {
        // faire un traitement...
      }
    }
  }
}
```

L'attribut System.Runtime.Remoting.Contexts. SynchronizationAttribute

Lorsque l'attribut `System.Runtime.Remoting.Contexts.Synchronization` est appliqué à une classe, une instance de cette classe ne peut pas être accédée par plus d'un thread à la fois.

Pour que ce comportement s'applique correctement il faut que la classe sur laquelle s'applique l'attribut soit *context-bound*, c'est-à-dire qu'elle doit dériver de la classe `System.ContextBoundObject`. La signification du terme context-bound est expliquée page 692.

Voici un exemple illustrant comment appliquer ce comportement :

Exemple 5-14.

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Threading;

[Synchronization(SynchronizationAttribute.REQUIRED)]
public class UneClasse : ContextBoundObject
{
  public void AfficheThreadHashVal()
  {
    Console.WriteLine("Début: ThreadHashVal= " +
          Thread.CurrentThread.GetHashCode() );
    Thread.Sleep(1000);
    Console.WriteLine("Fin:   ThreadHashVal= " +
          Thread.CurrentThread.GetHashCode() );
```

CS_doNet/05 Page 147 Mercredi, 14. mai 2003 10:32 10


```
  }
}

public class Prog
{
  static UneClasse m_Objet;
  static void Main()
  {
    m_Objet = new UneClasse();
    Thread T0 = new Thread(new ThreadStart(ThreadProc));
    Thread T1 = new Thread(new ThreadStart(ThreadProc));
    T0.Start(); T1.Start();
    T0.Join(); T1.Join();
  }
  static void ThreadProc()
  {
    for(int i=0;i<2;i++)
      m_Objet.AfficheThreadHashVal();
  }
}
```

Cet exemple affiche :

```
Début: ThreadHashVal= 27
Fin:   ThreadHashVal= 27
Début: ThreadHashVal= 28
Fin:   ThreadHashVal= 28
Début: ThreadHashVal= 27
Fin:   ThreadHashVal= 27
Début: ThreadHashVal= 28
Fin:   ThreadHashVal= 28
```

Notez que le Framework .NET présente un autre attribut ayant ce nom mais faisant partie d'une autre espace de noms. Cet attribut `System.EntrepriseServices.Synchronization` a la même finalité, mais il utilise le service d'entreprise COM+ de synchronisation. Cependant l'utilisation de l'attribut `System.Runtime.Remoting.Contexts.Synchronization` est préférable pour deux raisons :

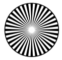
- Son utilisation est plus performante.
- Ce mécanisme supporte les appels asynchrones, contrairement à la version COM+.

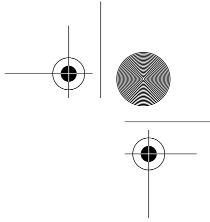
La notion de service d'entreprise COM+ est présentée page 595.

Gestion d'un pool de threads

Introduction

Le concept de *pool de threads* n'est pas nouveau. Cependant le Framework .NET vous permet d'utiliser un pool de threads beaucoup plus simplement que n'importe quelle autre technologie, grâce à la classe `System.Threading.ThreadPool`.





CS_doNet/05 Page 148 Mercredi, 14. mai 2003 10:32 10

Dans une application multithreads, la plupart des threads passent leur temps à attendre des événements. Concrètement vos threads sont globalement sous exploités. De plus, le fait de devoir tenir compte de la gestion des threads lors du design de votre application est une difficulté dont on se passerait volontiers.

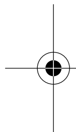
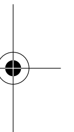
L'utilisation d'un pool de threads résout de façon élégante et performante ces problèmes. Vous postez des tâches au pool qui se charge de les distribuer à ces threads. Le pool est entièrement responsable de :

- la création et de la destruction de ses threads ;
- la distribution des tâches ;
- l'utilisation optimale de ses threads.

Le développeur est donc déchargé de ces responsabilités.

Malgré tous ces avantages, il est souhaitable de ne pas utiliser de pool de threads lorsque :

- Vos tâches doivent être gérées avec un système de priorité ;
- Vos tâches sont longues à s'exécuter (plusieurs secondes) ;
- Vos tâches doivent être traitées dans des *STA* (*Single Apartment Thread*). En effet les threads d'un pool sont de type *MTA* (*Multiple Apartement Thread*). Cette notion de *thread apartment*, inhérente à la technologie COM, est expliquée page 238.

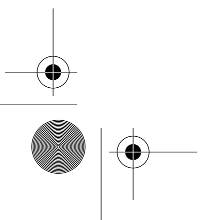
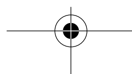
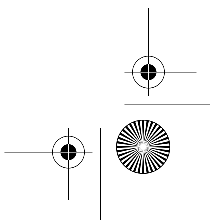
Utilisation d'un pool de threads

En .NET il n'y a qu'un pool de threads par processus. Ainsi toutes les méthodes de la classe `ThreadPool` sont statiques puisqu'elles s'appliquent à l'unique pool. Le Framework .NET utilise ce pool pour les appels asynchrones, décrit un peu plus loin page 151, les mécanismes d'*entré/sortie* asynchrones (aussi appelés *completion ports* en anglais, ils sont décrit page 489 et page 496) ou les timers.

Le nombre maximal de threads du pool est de 25 threads par processeur pour traiter les opérations asynchrones et de 25 *threads ouvrier* (*worker thread* en anglais) par processeur. Ces deux limites par défaut sont modifiables à tout moment en appelant la méthode `CorSetMaxThreads` de l'interface COM `ICorThreadpool` définie dans la DLL cale `mscoree.dll`. Pour accéder à cette fonctionnalité, il faut fabriquer son propre hôte du moteur d'exécution, comme expliqué page 81. Si le nombre maximal de threads est atteint dans le pool, ce dernier ne crée plus de nouveaux threads et les tâches dans la file du pool ne seront traitées que lorsqu'un thread du pool se libèrera. En revanche, le thread responsable de la création de la tâche, n'a pas à attendre qu'elle soit traitée.

Vous pouvez utiliser le pool de threads de deux façons différentes :

- En postant vos propres tâches et leurs méthodes de traitement avec la méthode `QueueUserWorkItem()`. Une fois qu'une tâche a été postée au pool, elle ne peut plus être annulée.
- En créant un timer qui poste périodiquement une tâche prédéfinie et sa méthode de traitement au pool. Pour cela, il faut utiliser la méthode `RegisterWaitForSingleObject()`.

CS_doNet/05 Page 149 Mercredi, 14. mai 2003 10:32 10

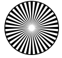
Notez que chacune de ces deux méthodes existe dans une version non protégée (`UnsafeQueueUserWorkItem()` et `UnsafeRegisterWaitForSingleObject()`). Ces versions non protégées permettent aux threads ouvriers du pool de ne pas être dans le même contexte de sécurité que le thread qui a déposé la tâche. De plus l'utilisation de ces méthodes améliore les performances puisque la pile du thread qui a déposé la tâche n'est pas vérifiée lors de la gestion des contextes de sécurité.

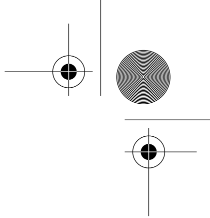
Notez que les méthodes de traitement sont référencées par des *délégués*. Cette notion de délégué est présentée page 401.

Voici un exemple qui montre l'utilisation de tâches utilisateurs (paramétrées par un numéro et traitées par la méthode `ThreadTache()`) et de tâches postées périodiquement (sans paramètre et traitées par la méthode `ThreadTacheWait()`). Les tâches utilisateurs sont volontairement longues pour forcer le pool à créer de nouveaux threads. Lorsque l'on utilise un pool de threads, il vaut mieux ne pas pouvoir différencier les threads du pool. Cependant, pour les besoins de l'exemple les threads du pool sont nommés à leur création dans la méthode `NommeEventuellementCeThread()` pour pouvoir observer quel thread exécute quelle tâche :

Exemple 5-15.

```
using System;
using System.Threading;

public class ThreadPoolTest
{
  public static int Main(string[] args)
  {
    // Positionnement initial de l'événement: false
    AutoResetEvent ev = new AutoResetEvent(false);

    ThreadPool.RegisterWaitForSingleObject(
      ev,
          // Méthode de traitement de la tâche périodique
      new WaitOrTimerCallback(ThreadTacheWait),
      null, // la tâche périodique n'a pas de paramètre
      2000, // la période est de 2 secondes
      false
      );

    // vous avez la possibilité de poster une tâche périodique
    // en appelant 'ev.Set()'

    // Poste 5 tâches utilisateur de paramètres 1,2,3,4,5
    for(int count = 0; count < 5; ++count)
      ThreadPool.QueueUserWorkItem(
        new WaitCallback(ThreadTache), count);
        // Attente de 20 secondes avant de finir le processus
    // car les threads du pool sont des threads background
    Thread.Sleep(20000);
    return 0;
  }
```





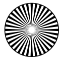
CS_doNet/05 Page 150 Mercredi, 14. mai 2003 10:32 10

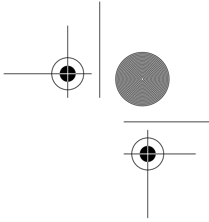

```
  static void ThreadTache(Object Obj)
  {
    NommeEventuellementCeThread();
    Console.WriteLine("{0}  Tâche#{1}  Début",
          Thread.CurrentThread.Name,Obj.ToString());
    Thread.Sleep(5000);
    Console.WriteLine("{0}  Tâche#{1}  Fin",
          Thread.CurrentThread.Name,Obj.ToString());
  }

  static void ThreadTacheWait(object Obj, bool signaled)
  {
    NommeEventuellementCeThread();
    Console.WriteLine("{0}  TâcheWait",  Thread.CurrentThread.Name);
  }

  static int ThreadNumber = 0;
  static void NommeEventuellementCeThread()
  {
    if( Thread.CurrentThread.Name == null )
    {
      Thread.CurrentThread.Name = "Thread#"+ThreadNumber;
      // Synchronise l'incrémentation de ThreadNumber
      Interlocked.Increment( ref ThreadNumber);
    }
  }
}
```

Ce programme affiche ceci (d'une manière non déterministe):

```
Thread#0  Tâche#0  Début
Thread#1  Tâche#1  Début
Thread#2  Tâche#2  Début
Thread#0  Tâche#0  Fin
Thread#0  Tâche#3  Début
Thread#1  Tâche#1  Fin
Thread#1  Tâche#4  Début
Thread#2  Tâche#2  Fin
Thread#2  TâcheWait
Thread#2  TâcheWait
Thread#2  TâcheWait
Thread#2  TâcheWait
Thread#2  TâcheWait
Thread#0  Tâche#3  Fin
Thread#1  Tâche#4  Fin
Thread#2  TâcheWait
Thread#0  TâcheWait
Thread#0  TâcheWait
Thread#0  TâcheWait
Press any key to continue
```

CS_doNet/05 Page 151 Mercredi, 14. mai 2003 10:32 10





Appel asynchrone d'une méthode

Pour aborder cette section, il est nécessaire d'avoir compris la notion de délégué, expliquée page 401.

On dit d'un appel de méthode qu'il est synchrone lorsque le thread du coté qui réalise l'appel, attend que la méthode soit exécutée avant de continuer. Ce comportement consomme des ressources puisque le thread est bloqué pendant ce temps. Lors d'un appel sur un objet distant, cette durée est potentiellement immense, puisque le coût d'un appel réseau représente des milliers, voire des millions, de cycles processeurs. Cependant cette attente n'est obligatoire que dans le cas où les informations retournées par l'appel de la méthode sont immédiatement consommées après l'appel. Dans la programmation en général et dans les architectures distribuées en particulier, il arrive souvent qu'un appel de méthode effectue une action et ne retourne que l'information décrivant si l'action s'est bien passée ou non. Dans ce cas, le programme n'a pas forcément besoin de savoir immédiatement si l'action s'est bien passée. On peut décider d'essayer de recommencer cette action plus tard si on apprend qu'elle a échoué.

Pour gérer ce type de situation on peut utiliser un *appel asynchrone*. L'idée est que le thread qui réalise un appel de méthode sur un objet, retourne immédiatement, sans attendre la fin de l'appel. L'appel est automatiquement pris en charge par un thread du pool de threads du processus. Le programme peut ultérieurement récupérer les informations retournées par l'appel asynchrone. La technique d'appel asynchrone est entièrement gérée par le CLR.

Le mécanisme que nous décrivons peut être utilisé dans votre propre architecture. Il est aussi utilisé par les classes du Framework .NET, notamment pour gérer les flots de données d'une manière asynchrone ou pour gérer des appels asynchrones sur des objets distants, c'est-à-dire situés dans un autre domaine d'application.

Délégation asynchrone

Lors d'un appel asynchrone vous n'avez pas à créer ni à vous occupez du thread qui exécute le corps de la méthode. Ce thread est géré par le pool de threads décrit un peu plus haut.

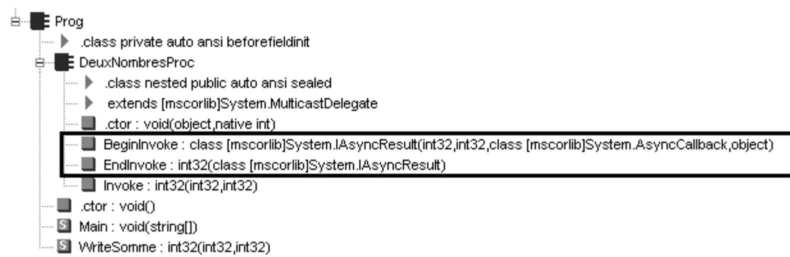
Avant d'utiliser effectivement un *délégué asynchrone*, il est judicieux de remarquer que lorsque vous compilez le code de l'Exemple 5-16, la classe déléguée `DeuxNombresProc` implémente les méthodes `BeginInvoke()` et `EndInvoke()`, générées par le compilateur. Pour s'en convaincre, la représente l'analyse de l'assemblage créé avec `ildasm.exe`. Je fais cette remarque car le mécanisme d'intellisense de Visual Studio .NET ne connaît pas ces méthodes : elles sont produites à la compilation.

CS_doNet/05 Page 152 Mercredi, 14. mai 2003 10:32 10

Exemple 5-16.

```
using System;
using System.Threading;

class Prog
{
  public delegate int DeuxNombresProc(int a,int b);
  static int WriteSomme(int a,int b)
  {
    Console.WriteLine(
       "Somme = {0}" ,a+b);
    return a+b;
  }
  static void Main(string[] args)
  {
    DeuxNombresProc Proc = new DeuxNombresProc(WriteSomme);
    Proc(10,10) ;
  }
}
```

Figure 5-4. Les méthodes BeginInvoke() et EndInvoke() d'un délégué



Pour appeler une méthode d'une manière asynchrone, il suffit de référencer la méthode dans un délégué avec la même signature que la méthode, et d'appeler la méthode `BeginInvoke()` sur ce délégué. Comme vous l'avez remarqué, le compilateur a fait en sorte que les premiers arguments de `BeginInvoke()` soient les arguments de la méthode à appeler. Les deux derniers arguments de cette méthode font l'objet des prochaines sections.

La valeur de retour de l'appel asynchrone d'une méthode, peut être récupérée en appelant la méthode `EndInvoke()`. Là aussi, le compilateur a fait en sorte que le type de la valeur de retour de `EndInvoke()` soit le même que le type de la valeur de retour de la délégation (ce type est int dans notre exemple). L'appel à `EndInvoke()` est bloquant. C'est-à-dire que l'appel ne retourne que lorsque l'exécution asynchrone est effectivement terminée.

Le code de l'exemple précédent a été modifié, de façon que la méthode `WriteSomme()` soit appelée d'une manière asynchrone. Notez que pour bien différencier le thread exécutant la méthode `Main()` et le thread exécutant la méthode `WriteSomme()`, nous affichons la valeur de hachage du thread courant (qui est différente pour chaque thread).

CS_doNet/05 Page 153 Mercredi, 14. mai 2003 10:32 10

Exemple 5-17.

```
using System;
using System.Threading;

class Prog
{
  public delegate int DeuxNombresProc(int a,int b);
  static int WriteSomme(int a,int b)
  {
    int Somme = a+b;
    Console.WriteLine("Thread {0}:WriteSomme() Somme = {1}",
           Thread.CurrentThread.GetHashCode(),Somme);
    return Somme;
  }
  static void Main(string[] args)
  {
    DeuxNombresProc Proc = new DeuxNombresProc(WriteSomme);
    IAsyncResult Async  = Proc.BeginInvoke(10,10,null,null);
    int Somme = Proc.EndInvoke(Async);
    Console.WriteLine("Thread {0}:Main()     Somme = {1}",
           Thread.CurrentThread.GetHashCode(),Somme);
  }
}
```

Sur ma machine, ce programme affiche :

```
Thread 15:WriteSomme() Somme = 20
Thread 18:Main()       Somme = 20
```

Un appel asynchrone est matérialisé par un objet dont la classe implémente l'interface `System.IAsyncResult`. Dans cet exemple la classe sous-jacente est `System.Runtime.Remoting.Messaging.AsyncResult`. L'objet `AsyncResult` est retourné par la méthode `BeginInvoke()`. Il est passé en argument de la méthode `EndInvoke()` pour identifier l'appel asynchrone.

Notez que si une exception est lancée lors d'un appel asynchrone, elle est automatiquement interceptée et stockée par le CLR. Le CLR relancera l'exception lors de l'appel à `EndInvoke()`.

Procédure de finalisation

Vous avez la possibilité de spécifier une méthode qui sera automatiquement appelée lorsque l'appel asynchrone sera terminé. Cette méthode est appelée *procédure de finalisation*. Une procédure de finalisation est appelée par le même thread que celui qui a exécuté l'appel asynchrone.

Pour utiliser une procédure de finalisation, il vous suffit de spécifier la méthode dans une délégation de type `System.AsyncCallback` comme avant-dernier paramètre de la méthode `BeginInvoke()`. Cette méthode doit être conforme à cette délégation, c'est-à-dire qu'elle doit retourner le type void et prendre pour seul argument une interface `IAsyncResult`. Comme le montre l'exemple suivant, cette méthode doit appeler `EndInvoke()`.

CS_doNet/05 Page 154 Mercredi, 14. mai 2003 10:32 10

Un problème se pose, car les threads du pool utilisés pour traiter les appels asynchrones sont des threads background. A l'instar de l'exemple ci-dessous, il faut que vous implémentiez un mécanisme de gestion d'événements pour vous assurer que l'application ne se termine pas sans avoir terminé l'exécution asynchrone. Notez que l'interface `IAsyncResult` présente un objet de synchronisation d'une classe dérivée de `WaitHandle`, mais cet objet est signalé dès que le traitement asynchrone est fini et avant que la procédure de finalisation soit appelée. Cet objet ne peut donc pas permettre d'attendre la fin de l'exécution de la procédure de finalisation.

Exemple 5-18.

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

class Prog
{
  public delegate int DeuxNombresProc(int a,int b);

  // position initiale de l'événement : false
  static AutoResetEvent E = new AutoResetEvent(false);
  static int WriteSomme(int a,int b)
  {
    Console.WriteLine(
      "{0}: Somme = {1}",Thread.CurrentThread.GetHashCode(),a+b);
    return a+b;
  }
  static void SommeFinie(IAsyncResult Async)
  {
    // attend une seconde pour simuler un traitement long
    Thread.Sleep(1000);
    DeuxNombresProc Proc = (DeuxNombresProc)
        ((AsyncResult)Async).AsyncDelegate;
    int Somme = Proc.EndInvoke(Async);
    Console.WriteLine(
        "{0}: Procédure de finalisation Somme = {1}",
        Thread.CurrentThread.GetHashCode(),Somme);

    E.Set();
  }
  static void Main(string[] args)
  {
    DeuxNombresProc Proc = new DeuxNombresProc(WriteSomme);
    IAsyncResult Async = Proc.BeginInvoke(10,10,
      new AsyncCallback(SommeFinie),null);

    Console.WriteLine(
     "{0}: BeginInvoke() appelée! Attend l'éxécution de SommeFinie() ",
     Thread.CurrentThread.GetHashCode());
```

CS_doNet/05 Page 155 Mercredi, 14. mai 2003 10:32 10


```
    E.WaitOne();
    Console.WriteLine(
        "{0}: Bye... ",Thread.CurrentThread.GetHashCode());
  }
}
```

Cet exemple affiche :

```
12: BeginInvoke() appelée! Attend l'éxécution de SommeFinie()
14: Somme = 20
14: Procédure de finalisation Somme = 20
12: Bye...
```

Si vous enlevez le mécanisme d'événement, cet exemple affiche ceci :

```
12: BeginInvoke() appelée! Attend l'éxécution de SommeFinie()
12: Bye...
```

L'application n'attend pas la fin de l'exécution du traitement asynchrone et de sa procédure de finalisation.

Passage d'un état à la procédure de finalisation

Si vous ne le positionnez pas à `null`, le dernier paramètre de la méthode `BeginInvoke()` représente une référence vers un objet utilisable à la fois dans le thread qui déclenche l'appel asynchrone et dans la procédure de finalisation. Une autre référence vers cet objet est la propriété `AsyncState` de l'interface `IAsyncResult`. Vous pouvez vous en servir pour représenter un état, positionné dans la procédure de finalisation. Par exemple, l'événement de l'exemple de la section précédente peut être vu comme un état. Voici le code réécrit pour utiliser cette fonctionnalité :

Exemple 5-19.

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

class Prog
{
  public delegate int DeuxNombresProc(int a,int b);

  static int WriteSomme(int a,int b)
  {
    int Somme = a+b;
    Console.WriteLine(
       "{0}: Somme = {1}",Thread.CurrentThread.GetHashCode(),Somme);
    return Somme;
  }
  static void SommeFinie(IAsyncResult Async)
  {
```

CS_doNet/05 Page 156 Mercredi, 14. mai 2003 10:32 10


```
    // Attend une seconde pour simuler un traitement long.
    Thread.Sleep(1000);
    DeuxNombresProc Proc = (DeuxNombresProc)
      ((AsyncResult)Async).AsyncDelegate;
    int Somme = Proc.EndInvoke(Async);
    Console.WriteLine(
      "{0}: Procédure de finalisation Somme = {1}",
      Thread.CurrentThread.GetHashCode(),Somme);

    ((AutoResetEvent)Async.AsyncState).Set();
  }
  static void Main(string[] args)
  {
    DeuxNombresProc Proc = new DeuxNombresProc(WriteSomme);
    AutoResetEvent E = new AutoResetEvent(false);

    IAsyncResult Async = Proc.BeginInvoke(10,10,
      new AsyncCallback(SommeFinie), E );

    Console.WriteLine(
    "{0}: BeginInvoke() appelée! Attend l'éxécution de SommeFinie() ",
      Thread.CurrentThread.GetHashCode());

    E.WaitOne();
    Console.WriteLine(
      "{0}: Bye... ",Thread.CurrentThread.GetHashCode());
  }
}
```

Appels sans retour (One-Way)

Vous avez la possibilité d'appliquer l'attribut `System.Runtime.Remoting.Messaging.OneWay` à n'importe quelle méthode, statique ou non. Cet attribut indique au CLR que cette méthode ne retourne aucune information. Même si une méthode qui retourne une valeur de retour ou des arguments de retour (i.e définis avec le mot-clé `out`) est marquée avec cet attribut, elle ne retourne rien.

Une méthode marquée avec l'attribut `OneWay` peut être appelée d'une manière synchrone ou asynchrone. Si une exception est lancée et non rattrapée durant l'exécution d'une méthode marquée avec l'attribut `OneWay`, elle est propagée si l'appel est synchrone. Dans le cas d'un appel asynchrone sans retour l'exception n'est pas propagée. Dans la plupart des cas, les méthodes marquées sans retour sont appelées de manière asynchrone.

Les appels asynchrones sans retour effectuent en général des tâches annexes dont la réussite ou l'échec n'ont pas d'incidence sur le bon déroulement de l'application. La plupart du temps, on les utilise pour communiquer des informations sur le déroulement de l'application.