

## Petit cas d'illustration

Rien de tel qu'une belle métaphore pour expliquer ce concept.

Imaginez que vous allez vous marier, et c'est le moment d'envoyer les invitations. Si vous êtes déjà marié, revenez quelques années en arrière à ce même moment. Et, si vous l'avez fait organiser par quelqu'un d'autre, imaginez que vous l'avez fait vous-même.

Comme vous avez énormément d'amis, il y a environ un millier de lettres à envoyer. Vous vous munissez des enveloppes, des adresses, des timbres, et avec beaucoup de bonne volonté vous vous y mettez. Au bout d'un certain temps (qui pour moi serait inenvisageable, je dois l'avouer, je n'ai pas beaucoup de patience), finalement, vous aurez réussi à envoyer toutes les lettres.

Maintenant, imaginez le même travail, sauf que vous avez un ami pour vous aider. Vous lui passez la moitié des enveloppes, la moitié des adresses, vous partagez les timbres, vous prenez bien soin de lui expliquer ce qu'il faut faire et vous vous y mettez à deux. Outre le fait que ce sera plus marrant et plus sympa, vous allez surtout mettre deux fois moins de temps.

Pareil, mais avec vingt amis. Vous mettez vingt fois moins de temps. En revanche, si on essayait avec cinq cents, on pourrait penser que cela prendrait cinq cents fois moins de temps mais, finalement, vous vous rendez compte que le temps d'expliquer à chacune des cinq cents personnes ce qu'il faut faire, vous auriez pu envoyer au moins cinq mille lettres.

Il y a donc un nombre d'amis qui optimise le temps d'envoi de toutes vos lettres.

Le multithreading, c'est exactement cela.

- Les lettres à envoyer représentent le but de votre programme, ce qu'il doit faire.
- Les enveloppes, les adresses et les timbres représentent les données.
- Et vous et vos amis êtes des threads. (Ne vous inquiétez pas, on s'y fait bien !)

Mais vous avez vu qu'il a fallu expliquer à vos amis ce qu'il fallait faire. De la même manière, il vous faudra préparer les threads aux actions qu'ils devront effectuer.

De plus, vous avez dû partager les enveloppes et les timbres. Pareillement, vous aurez à vous arranger pour partager les données entre les threads.

En revanche, il n'est pas impossible qu'un de vos amis ait utilisé vos timbres parce qu'il n'en avait plus ou alors qu'il ait fini bien avant les autres. Cela peut arriver également au niveau des threads. Ils ne s'arrêteront pas en même temps et pourront être amenés à utiliser les mêmes données. Il s'agit là de la problématique principale du multithreading : l'accès concurrent aux données.

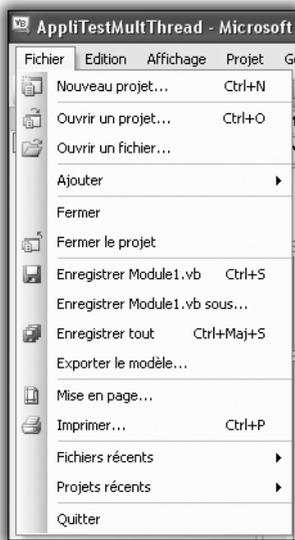
En effet, la principale source d'erreur lorsque l'on traite du multithreading est due à des données modifiées par des threads alors qu'un autre thread s'en servait.

Imaginez que vous vouliez enregistrer une émission à la télévision. Vous programmez l'enregistrement sur une chaîne donnée. Votre conjoint passe par là et change de chaîne pour regarder son émission. Et vous vous retrouvez avec le programme de cette chaîne plutôt que sur celui que vous vouliez enregistrer. Le problème a été posé par un accès concurrent à la télévision. Vous avez tous les deux voulu l'utiliser et le traitement de l'un a perturbé l'autre.

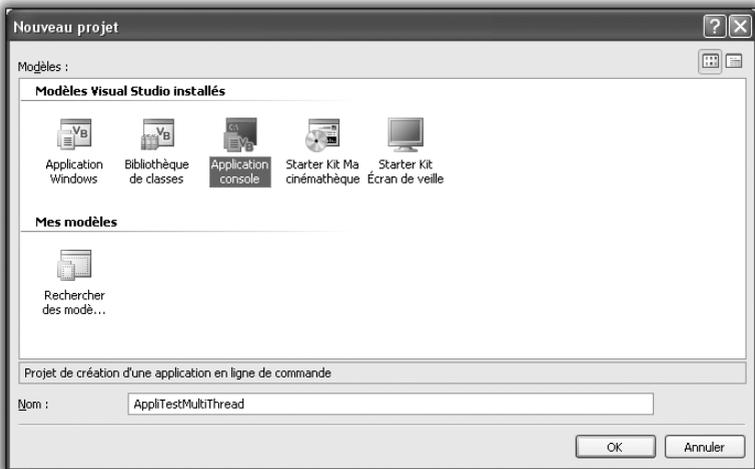
Maintenant que nous avons vu la théorie sur ce qu'est le multithreading, faisons une application pour illustrer concrètement ce qui se passe.

## Application explicative

- 1 Tout d'abord, créons un nouveau projet (voir Figure 9.76).
- 2 On l'appellera `AppliTestMultiThread`. Ce sera une application Console simple pour présenter le concept de multithreading (voir Figure 9.77).



**Figure 9.76 :**  
*Création d'un nouveau projet*



**Figure 9.77 :** *Nouvelle application Console.*

Dans un premier temps, notre application ne fera que boucler sur une valeur et l'écrire. Elle marquera ensuite la fin du programme, que nous finirons par un `ReadLine` pour que la fenêtre ne disparaisse pas tout de suite.

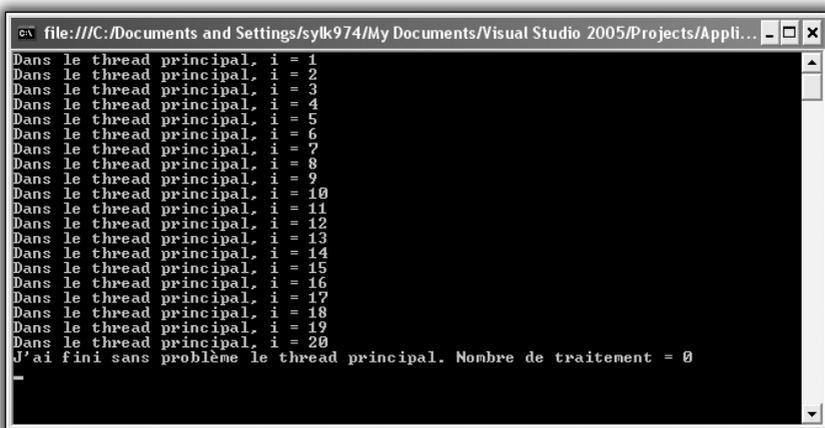
```
Sub Main()  
    Dim count As Integer = 0
```

```
For i = 1 To 20
    ' Traitement à faire 100X
    'count += 1
    Console.WriteLine("Dans le thread principal, i = "
        & i.ToString)
Next i

Console.WriteLine("J'ai fini sans problème le thread
    principal. Nombre de traitement = "
        & count.ToString)

Console.ReadLine()
End Sub
```

L'exécution de ce programme devrait vous ressortir ceci :



```
file:///C:/Documents and Settings/sylk974/My Documents/Visual Studio 2005/Projects/Appi...
Dans le thread principal, i = 1
Dans le thread principal, i = 2
Dans le thread principal, i = 3
Dans le thread principal, i = 4
Dans le thread principal, i = 5
Dans le thread principal, i = 6
Dans le thread principal, i = 7
Dans le thread principal, i = 8
Dans le thread principal, i = 9
Dans le thread principal, i = 10
Dans le thread principal, i = 11
Dans le thread principal, i = 12
Dans le thread principal, i = 13
Dans le thread principal, i = 14
Dans le thread principal, i = 15
Dans le thread principal, i = 16
Dans le thread principal, i = 17
Dans le thread principal, i = 18
Dans le thread principal, i = 19
Dans le thread principal, i = 20
J'ai fini sans problème le thread principal. Nombre de traitement = 0
-
```

**Figure 9.78** : Exécution du programme

Utilisons maintenant dans ce même programme un autre thread qui va faire la même chose. Pour cela, il faut déclarer une méthode qui sera le point d'entrée du thread, c'est-à-dire la première qu'il va exécuter. Dans notre cas, on aura une méthode qui fera la même boucle que celle du programme principal en précisant que c'est celle du thread secondaire.

```
Sub MakeLoop()
    Dim count As Integer = 0

    For i = 1 To 20
        count += 1
        Console.WriteLine("Dans le thread secondaire, i = "
            & i.ToString)
    Next i
```

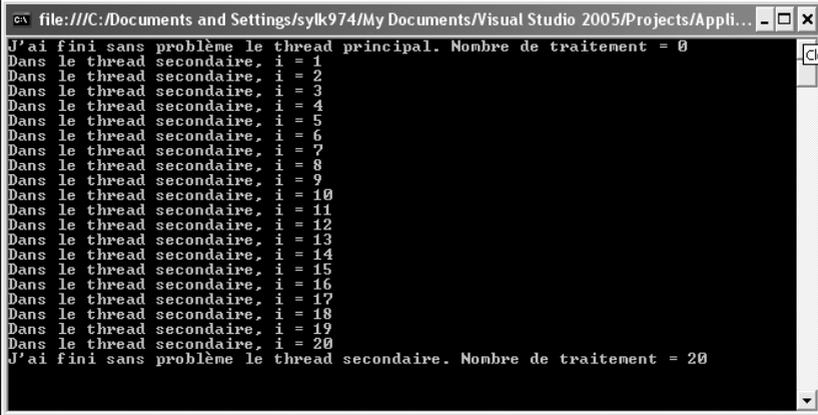
```
Console.WriteLine("J'ai fini sans problème le thread  
secondaire. Nombre de traitement = "  
& count.ToString)
```

```
End Sub
```

Maintenant, pour utiliser un nouveau thread, il suffit tout simplement de l'instancier en précisant son point de départ, ici notre méthode `MakeLoop`. Il faut pour cela utiliser le mot-clé `AddressOf` suivi du nom de la méthode qui sera le point d'entrée. La classe de `Thread` est dans le namespace `Threading`. Ensuite, pour exécuter le thread il n'y a plus qu'à faire `Start`.

```
Imports System.Threading  
Sub Main()  
    Dim count As Integer = 0  
    Dim threadSecondaire As New Thread(AddressOf MakeLoop)  
    threadSecondaire.Start()  
  
    Console.WriteLine("J'ai fini sans problème le thread  
principal. Nombre de traitement = "  
        & count.ToString)  
    Console.ReadLine()  
End Sub
```

À l'exécution, nous obtiendrons ceci :



```
file:///C:/Documents and Settings/sylk974/My Documents/Visual Studio 2005/Projects/Appli...  
J'ai fini sans problème le thread principal. Nombre de traitement = 0  
Dans le thread secondaire, i = 1  
Dans le thread secondaire, i = 2  
Dans le thread secondaire, i = 3  
Dans le thread secondaire, i = 4  
Dans le thread secondaire, i = 5  
Dans le thread secondaire, i = 6  
Dans le thread secondaire, i = 7  
Dans le thread secondaire, i = 8  
Dans le thread secondaire, i = 9  
Dans le thread secondaire, i = 10  
Dans le thread secondaire, i = 11  
Dans le thread secondaire, i = 12  
Dans le thread secondaire, i = 13  
Dans le thread secondaire, i = 14  
Dans le thread secondaire, i = 15  
Dans le thread secondaire, i = 16  
Dans le thread secondaire, i = 17  
Dans le thread secondaire, i = 18  
Dans le thread secondaire, i = 19  
Dans le thread secondaire, i = 20  
J'ai fini sans problème le thread secondaire. Nombre de traitement = 20
```

**Figure 9.79** : Exécution dans un autre thread

Très bien, nous avons réussi à produire du code dans un autre thread. Mais nous remarquons une chose, c'est que la phrase de fin du thread principal a été écrite avant la première phrase du thread secondaire.

C'est tout à fait normal. Comme nous l'avons vu à la présentation du concept de thread, c'est un processus séparé. Le processus principal va donc continuer son exécution sans attendre le second thread. Celui-ci est bien exécuté en parallèle du premier.

Tout cela est très bien, mais ici on a juste déplacé le traitement. Il y a effectivement deux threads, mais il n'y en a qu'un seul qui travaille réellement. Remettons alors notre boucle d'origine dans le programme principal. Cependant, pour que l'exemple soit plus parlant, il va falloir ajouter une instruction dans les boucles, `Sleep`. Cette instruction permet de stopper le thread qui l'exécute pendant un instant donné, calculé en millisecondes. En faisant cela, on simule un traitement long, car en réalité faire une boucle de 20 est tellement rapide que l'on ne verra pas les subtilités du multithreading.

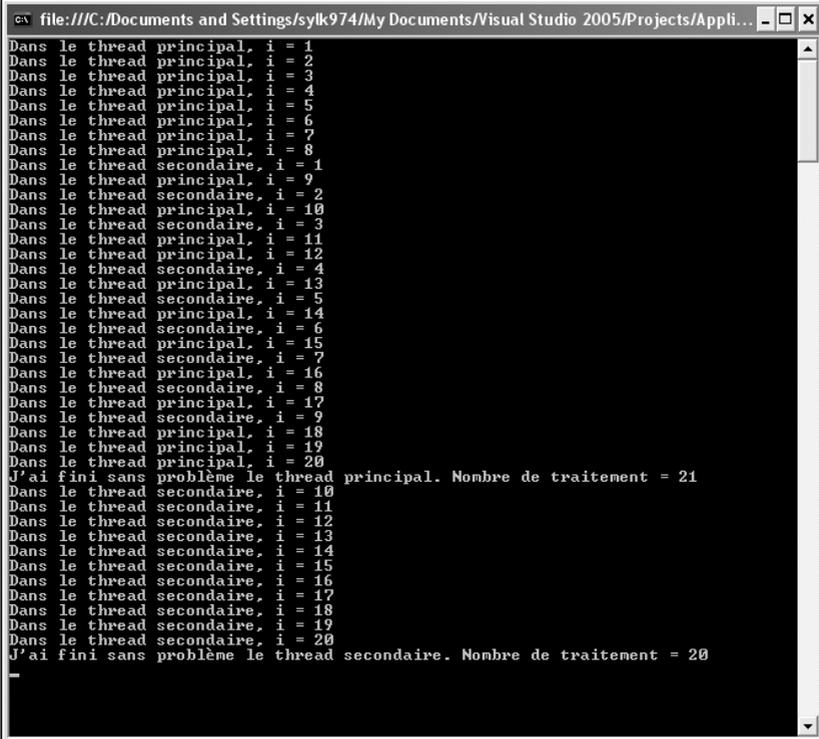
```
Sub Main()  
    Dim count As Integer = 0  
    Dim threadSecondaire As New Thread(AddressOf MakeLoop)  
    threadSecondaire.Start()  
  
    For count = 1 To 20  
        'count += 1  
        Thread.Sleep(0)  
        Console.WriteLine("Dans le thread principal, i = "  
            & count.ToString)  
    Next count  
  
    Console.WriteLine("J'ai fini sans problème le thread  
        principal. Nombre de traitement = "  
        & count.ToString)  
    Console.ReadLine()  
End Sub
```



### MakeLoop et Sleep

N'oubliez pas d'ajouter également l'instruction `Sleep` dans la méthode `MakeLoop`

Vous devriez obtenir un programme qui ressemble à ceci (la sortie peut varier d'un ordinateur à l'autre, car elle dépend de la vitesse de traitement de votre système d'exploitation) :



```
file:///C:/Documents and Settings/sytk974/My Documents/Visual Studio 2005/Projects/Appli...
Dans le thread principal, i = 1
Dans le thread principal, i = 2
Dans le thread principal, i = 3
Dans le thread principal, i = 4
Dans le thread principal, i = 5
Dans le thread principal, i = 6
Dans le thread principal, i = 7
Dans le thread principal, i = 8
Dans le thread secondaire, i = 1
Dans le thread principal, i = 9
Dans le thread secondaire, i = 2
Dans le thread principal, i = 10
Dans le thread secondaire, i = 3
Dans le thread principal, i = 11
Dans le thread principal, i = 12
Dans le thread secondaire, i = 4
Dans le thread principal, i = 13
Dans le thread secondaire, i = 5
Dans le thread principal, i = 14
Dans le thread secondaire, i = 6
Dans le thread principal, i = 15
Dans le thread secondaire, i = 7
Dans le thread principal, i = 16
Dans le thread secondaire, i = 8
Dans le thread principal, i = 17
Dans le thread secondaire, i = 9
Dans le thread principal, i = 18
Dans le thread principal, i = 19
Dans le thread principal, i = 20
J'ai fini sans problème le thread principal. Nombre de traitement = 21
Dans le thread secondaire, i = 10
Dans le thread secondaire, i = 11
Dans le thread secondaire, i = 12
Dans le thread secondaire, i = 13
Dans le thread secondaire, i = 14
Dans le thread secondaire, i = 15
Dans le thread secondaire, i = 16
Dans le thread secondaire, i = 17
Dans le thread secondaire, i = 18
Dans le thread secondaire, i = 19
Dans le thread secondaire, i = 20
J'ai fini sans problème le thread secondaire. Nombre de traitement = 20
```

**Figure 9.80** : Votre première application multithread

Comme vous pouvez le constater, l'exécution dans l'un ou l'autre des deux threads est un peu aléatoire. Tantôt c'est le premier qui fait huit tours de boucle. Ensuite, ils alternent chacun leur tour. Puis le second thread finit, et donc le premier termine à son tour.

Jouez maintenant à modifier les valeurs de `Sleep` et le nombre de tours de boucle, vous verrez qu'utiliser deux threads est plus rapide que n'en utiliser qu'un seul pour le même traitement.

Tout se passe ici pour le mieux, on ne voit aucun problème, et pourtant nous avons optimisé notre temps de traitement, sans remettre en cause la pérennité de notre programme. Facile, il n'y a aucune donnée commune

utilisée par nos deux threads. Qu'à cela ne tienne, `i`, notre variable de boucle, va devenir une donnée globale qui sera commune à nos deux threads. (En vérité, ceci n'est pas recommandé, mais c'est pour vous faire toucher concrètement le problème d'accès concurrent aux données.)

```
Module Module1
    Dim i As Integer

    'réécrire le code de MakeLoop également

Sub Main()
    Dim count As Integer = 0
    Dim threadSecondaire As New Thread(AddressOf MakeLoop)
    threadSecondaire.Start()

    For i = 1 To 20
        count += 1
        Thread.Sleep(0)
        Console.WriteLine("Dans le thread principal, i = "
            & i.ToString)
    Next count

    Console.WriteLine("J'ai fini sans problème le thread
        principal. Nombre de traitement = "
            & count.ToString)

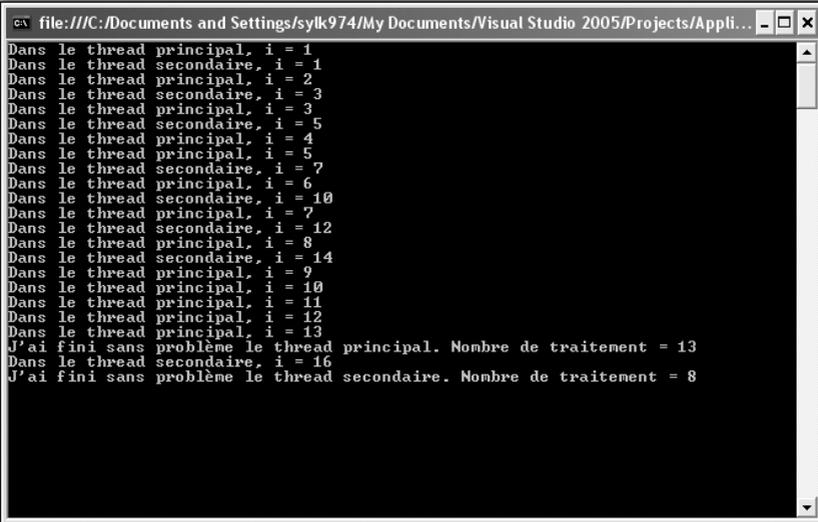
    Console.ReadLine()
Endd Sub

End Module
```

Et voici notre sortie : (voir Figure 9.81)

Et là, catastrophe... chacun de nos threads a fait la moitié de son travail. Au lieu d'avoir fait vingt opérations chacun, le premier en a fait treize, le second, huit, pas du tout ce qui était prévu à l'origine.

C'est là un problème d'accès concurrent aux données, mais nous allons voir que l'on peut y remédier.



```
file:///C:/Documents and Settings/sylk974/My Documents/Visual Studio 2005/Projects/Appli...
Dans le thread principal, i = 1
Dans le thread secondaire, i = 1
Dans le thread principal, i = 2
Dans le thread secondaire, i = 3
Dans le thread principal, i = 3
Dans le thread secondaire, i = 5
Dans le thread principal, i = 4
Dans le thread principal, i = 5
Dans le thread secondaire, i = 7
Dans le thread principal, i = 6
Dans le thread secondaire, i = 10
Dans le thread principal, i = 7
Dans le thread secondaire, i = 12
Dans le thread principal, i = 8
Dans le thread secondaire, i = 14
Dans le thread principal, i = 9
Dans le thread principal, i = 10
Dans le thread principal, i = 11
Dans le thread principal, i = 12
Dans le thread principal, i = 13
J'ai fini sans problème le thread principal. Nombre de traitement = 13
Dans le thread secondaire, i = 16
J'ai fini sans problème le thread secondaire. Nombre de traitement = 8
```

Figure 9.81 : Accès concurrent aux données

## Une solution naïve mais efficace : l'exclusion mutuelle

Dans notre programme précédent, le problème était dû à l'utilisation de la variable de boucle par deux threads différents. Donc, lorsqu'un des deux threads faisait un tour de boucle, il incrémentait cette variable et l'autre thread sautait un tour.

Comment régler ce problème ? La meilleure réponse possible : s'arranger pour qu'ils n'utilisent pas la même donnée. Mais comme il arrivera des cas où ce n'est pas possible (alors qu'ici on pourrait), nous allons chercher autre chose.

Comme j'aime bien les analogies, je vais tenter de vous faire toucher du doigt la solution par un cas de tous les jours : vous êtes au vidéoclub, vous voyez un film que vous voulez. Manque de chance, quelqu'un le prend avant vous. Que faites-vous ? Malheureusement, vous prenez votre mal en patience, et vous attendez que l'autre client le rapporte. L'analogie avec notre programme :

- Vous et l'autre client êtes des threads (vous vous y faites mieux la deuxième fois ?).
- La cassette représente la donnée partagée.
- Et notre solution, c'est l'attente. Cette solution se nomme l'exclusion mutuelle. Car, lorsqu'un thread accède à une donnée, il en bloque l'accès aux autres threads. De cette manière, il assure son traitement et rend ensuite la main.

Pour utiliser cette solution, il nous faut un verrou. Ce sera en fait un objet dans notre programme dont le seul but sera de dire aux threads *Attention c'est verrouillé, personne ne passe*. Une instance d'`Object` fera très bien l'affaire. Nous sommes obligés d'utiliser un objet supplémentaire car on ne peut pas utiliser comme verrou des objets qui sont amenés à changer.

```
Dim locker As Object = New Object()
```

Maintenant que nous avons un verrou, encore faut-il verrouiller la donnée quand on l'utilise. Ceci est fait avec le mot-clé `SynLock`, qui a comme argument l'objet verrou que l'on va utiliser. `SynLock` définit un bloc d'instructions qui seront toutes exécutées sans qu'aucune modification n'ait pu être faite sur le verrou.

```
Dim locker As Object = New Object()
```

```
Sub MakeLoop()  
    Dim count As Integer = 0  
  
    SyncLock locker  
  
        For i = 1 To 20  
            count += 1  
            Thread.Sleep(0)  
            Console.WriteLine("Dans le thread secondaire, i = "  
                & i.ToString)  
        Next i  
  
    End SyncLock  
  
    Console.WriteLine("J'ai fini sans problème le thread  
        secondaire. Nombre de traitement = "  
        & count.ToString)  
End Sub  
  
Sub Main()  
    Dim count As Integer = 0
```

```
Dim threadSecondaire As New Thread(AddressOf MakeLoop)
threadSecondaire.Start()
```

```
SyncLock locker
```

```
For i = 1 To 20
```

```
    count += 1
```

```
    Thread.Sleep(0)
```

```
    Console.WriteLine("Dans le thread principal, i = "
        & count.ToString)
```

```
Next i
```

```
End SyncLock
```

```
Console.WriteLine("J'ai fini sans problème le thread principal.
    Nombre de traitement = " & count.ToString)
```

```
Console.ReadLine()
```

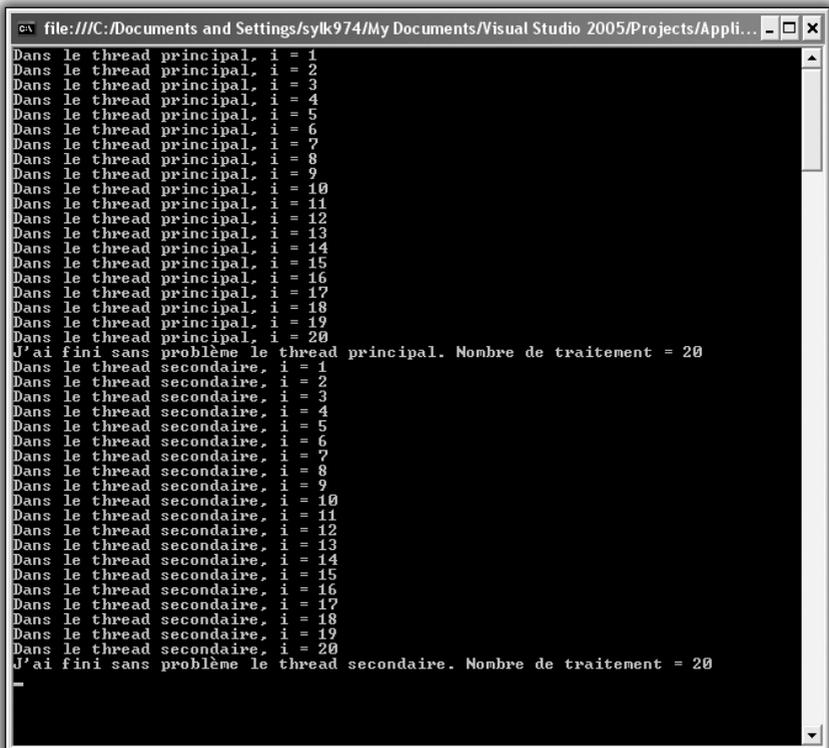
```
End Sub
```

Voici donc votre programme protégé des cas d'accès concurrent aux données. Le thread principal s'exécute. Il exécute le thread secondaire et continue son exécution. Il verrouille la donnée et entame sa boucle. De son côté, le thread secondaire, après s'être initialisé, tente de verrouiller la donnée. Or celle-ci a déjà été verrouillée par le thread principal, donc le thread secondaire attend. Le thread principal finit sa boucle et libère donc le verrou. Il finit ensuite son traitement. Le thread secondaire, qui attendait le verrou, voit celui-ci libéré. Il peut alors entamer l'exécution de sa boucle. C'est lui maintenant qui a mis le verrou. Il finit sa boucle. Fin du programme. Vous avez tout suivi ? Sinon voici l'illustration : (voir Figure 9.82)

En revanche, le revers de la médaille réside dans l'attente de libération du verrou. En effet, nous avons bien vu que, pendant le verrou, l'un des threads devait attendre l'autre. La performance est le prix de la sécurité.

De plus, dans de plus gros programmes, vous pouvez être amené à utiliser plusieurs verrous. Un problème bien connu dans l'utilisation des verrous est celui des verrous mortels : un thread 1 verrouille une donnée avec un verrou 1. Un autre thread 2 verrouille une autre donnée avec un verrou 2. Mais le thread 1 veut maintenant utiliser une donnée verrouillée par le verrou 2. Il attend donc. Et, de son côté, le thread 2 veut exécuter une instruction sur une donnée utilisée par le thread 1,

donc verrouillée par le verrou 1. Thread 1 attend Verrou 2 pour débloquent Verrou 1, mais Thread 2 attend Verrou 1 pour débloquent Verrou 2.



```
file:///C:/Documents and Settings/sylk974/My Documents/Visual Studio 2005/Projects/Appi...
Dans le thread principal, i = 1
Dans le thread principal, i = 2
Dans le thread principal, i = 3
Dans le thread principal, i = 4
Dans le thread principal, i = 5
Dans le thread principal, i = 6
Dans le thread principal, i = 7
Dans le thread principal, i = 8
Dans le thread principal, i = 9
Dans le thread principal, i = 10
Dans le thread principal, i = 11
Dans le thread principal, i = 12
Dans le thread principal, i = 13
Dans le thread principal, i = 14
Dans le thread principal, i = 15
Dans le thread principal, i = 16
Dans le thread principal, i = 17
Dans le thread principal, i = 18
Dans le thread principal, i = 19
Dans le thread principal, i = 20
J'ai fini sans problème le thread principal. Nombre de traitement = 20
Dans le thread secondaire, i = 1
Dans le thread secondaire, i = 2
Dans le thread secondaire, i = 3
Dans le thread secondaire, i = 4
Dans le thread secondaire, i = 5
Dans le thread secondaire, i = 6
Dans le thread secondaire, i = 7
Dans le thread secondaire, i = 8
Dans le thread secondaire, i = 9
Dans le thread secondaire, i = 10
Dans le thread secondaire, i = 11
Dans le thread secondaire, i = 12
Dans le thread secondaire, i = 13
Dans le thread secondaire, i = 14
Dans le thread secondaire, i = 15
Dans le thread secondaire, i = 16
Dans le thread secondaire, i = 17
Dans le thread secondaire, i = 18
Dans le thread secondaire, i = 19
Dans le thread secondaire, i = 20
J'ai fini sans problème le thread secondaire. Nombre de traitement = 20
-
```

**Figure 9.82** : Solution d'exclusion mutuelle

Ils ne pourront jamais se débloquent car l'un attend l'autre qui attend l'un (comme l'œuf et la poule) : ceci est un cas de verrou mortel, ou DeadLock.

Vous connaissez maintenant les principales possibilités du multithreading, ainsi que ses problématiques principales, mais aussi une solution pour y remédier. N'hésitez pas à pratiquer et à tester les moindres cas particuliers, car le multithreading est une notion bien subtile. Même des programmeurs professionnels font beaucoup d'erreurs lorsqu'ils introduisent des threads dans leurs applications. De plus, la nature aléatoire de l'exécution rend le débogage très difficile. La seule vraie technique pour maîtriser les threads, la curiosité et la

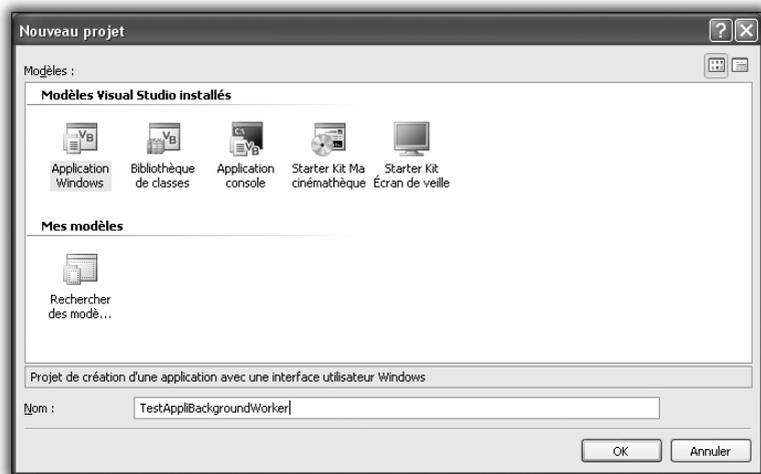
pratique : étudier la théorie pour comprendre les principes et les coder pour voir la réalité des choses. Bon courage !

## Le principe des tâches de fond

Contrôler des threads à la main offre bien des possibilités, mais on a également vu que cela pouvait poser pas mal de problèmes et obligeait à être particulièrement vigilant.

Heureusement, le framework .Net offre un mécanisme qui permet de bénéficier d'une capacité inhérente aux threads, l'exécution en tâche de fond. Vous pourrez de cette manière exécuter du code en arrière-plan et remonter des événements graphiques pour garder de l'interactivité dans votre programme.

- 1 Tout d'abord, créons un nouveau programme que nous appellerons `TestAppliBackgroundWorker`. Cette fois-ci, nous ferons une application graphique.



**Figure 9.83 :** Application de tâche de fond

- 2 Créez une interface graphique contenant au moins une *ProgressBar* et de quoi définir un nombre et lancer une action. Voici un modèle dont vous pourriez vous inspirer.



**Figure 9.84 :**  
Interface graphique

Créez maintenant une méthode qui va compter. Vous pourrez définir la limite haute en utilisant votre interface et vous indiquerez l'état d'avancement dans la `ProgressBar`.

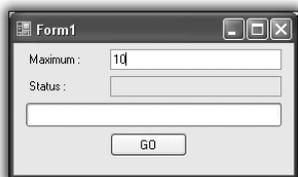
```
Public Sub Count()
    Dim count As Integer = Integer.Parse(TextBox1.Text)
    ProgressBar1.Maximum = count

    TextBox2.Text = "Launching count..."

    For i As Integer = 1 To count
        ProgressBar1.Value = i
        TextBox2.Text = "" & i.ToString()
    Next i

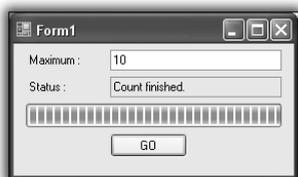
    TextBox2.Text = "Count finished."
End Sub
```

Maintenant, liez cette méthode à votre bouton, exécutez votre programme. Avec une petite limite haute, pas de souci :



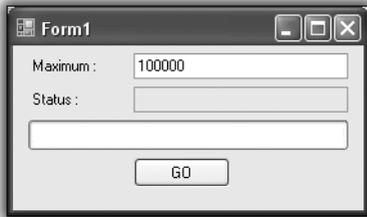
**Figure 9.85 :**  
Programme avec dix itérations

Le programme va très vite, mais on voit la `ProgressBar` défiler, le comptage se faire, et enfin on voit le message final.



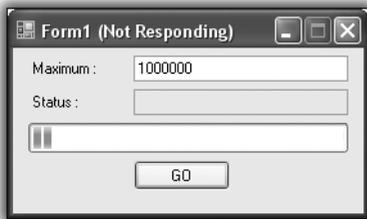
**Figure 9.86 :**  
Succès avec dix itérations

Jusqu'ici tout va bien. Que se passe-t-il avec un nombre beaucoup plus gros, comme 100 000 ou 1 million :



**Figure 9.87 :**  
*Programme avec un grand nombre*

Nous voyons la barre avancer un peu puis, au bout d'un moment, vous remarquerez qu'elle n'avance plus. Si à ce moment-là on essaie de cliquer un peu partout pour voir ce qui se passe, pas de réaction, mais un message *Not Responding* sur la fenêtre.



**Figure 9.88 :**  
*Blocage de l'application*

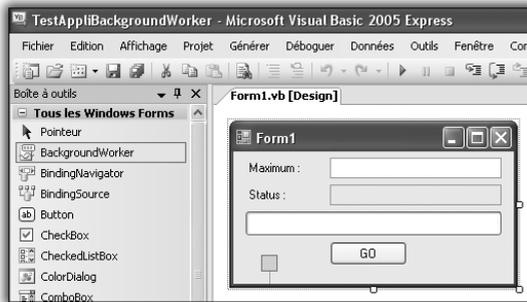
Quelle explication à cela ? Il se trouve que, si votre programme partage le même thread que votre élément graphique, tout traitement de votre programme, s'il est trop long, peut bloquer l'affichage de cet élément graphique. Vous reprendrez la main seulement quand il aura fini son traitement.

Dans notre cas, c'est le même thread qui a créé les éléments de l'interface graphique (le formulaire, le bouton, les labels...) et qui fait le traitement. En effet, nous n'avons pas encore inclus dans notre programme un quelconque mécanisme de multithreading. Lorsque nous avons testé avec 10 comme valeur, pas de souci car le traitement était assez rapide pour qu'on ne s'aperçoive pas du blocage de l'interface graphique. Mais, dès lors qu'on a mis une limite maximale plus grande, nous avons senti ce blocage. Nous avons pu voir le début de l'avancement, car le traitement n'était pas considéré comme assez grand pour bloquer l'affichage de l'interface graphique, mais au moment où il a dépassé cette limite il nous est devenu impossible de récupérer la

main. Ce sera possible en fin de traitement mais il est impossible d'en voir le déroulement.

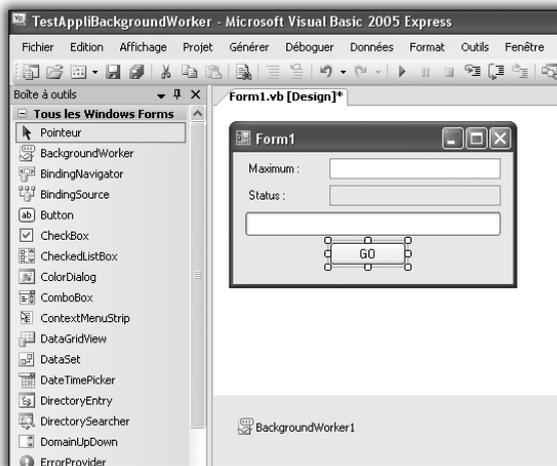
.Net propose un mécanisme un peu plus évolué qu'un thread, mais tout aussi simple à utiliser, qui nous permet d'exécuter du code en parallèle, de notifier de la progression et de finaliser l'exécution de celui-ci : le `BackgroundWorker`.

Pour ajouter un `BackgroundWorker` dans votre programme, il vous suffit de l'ajouter depuis la liste des contrôles vers votre interface.



**Figure 9.89 :**  
*Ajout d'un  
BackgroundWorker au  
programme*

Il apparaît maintenant dans votre programme, mais en dehors de votre interface graphique.



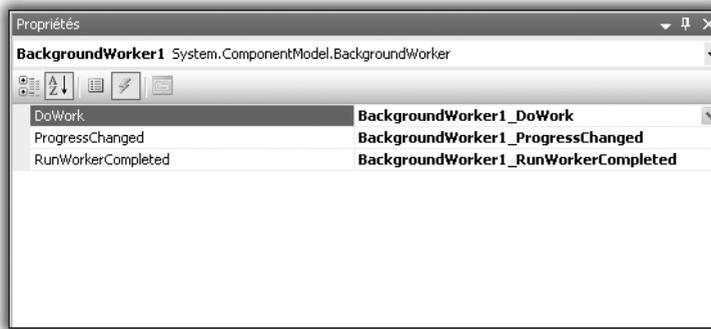
**Figure 9.90 :**  
*Intégration du  
BackgroundWorker*

Le `BackgroundWorker` se manie comme un objet normal, avec des propriétés, des attributs et des méthodes. La première chose à faire avec

le `BackgroundWorker` est de l'autoriser à notifier sa progression. Sans cette étape, il ne vous sera pas possible d'utiliser la méthode `ReportProgress`, qui permet de donner l'état d'avancement du `BackgroundWorker`.

```
BackgroundWorker1.WorkerReportsProgress = True
```

La première chose à définir est l'action à effectuer. Pour faire ceci, il faut traiter l'événement `DoWork` du `BackgroundWorker`. Il suffit d'aller dans l'écran des Propriétés et, dans la partie événement, de cliquer sur la partie `DoWork`.



**Figure 9.91** : Définition de l'action principale

En double-cliquant sur cet événement, Visual Studio vous crée le squelette de la méthode qui va faire l'action principale. Dans notre programme, l'action principale est de boucler sur une limite haute que vous aurez définie avant :

```
Public Sub BackgroundWorker1_DoWork(ByVal sender As
  System.Object, ByVal e As System.ComponentModel
  .DoWorkEventArgs) Handles BackgroundWorker1.DoWork
    Dim i As Integer
    For i = 0 To max
        counter = i
        BackgroundWorker1.ReportProgress(i * 100 /
  max)
    Next
End Sub
```

La limite haute, `max`, devra avoir été définie préalablement. Ceci se fera lors du `Click` sur le bouton. Nous verrons un peu plus loin le code de l'appel qui contient la création et le lancement du `BackgroundWorker`.

Nous remarquons une instruction dont je n'ai pas parlé : `ReportProgress`. Cette instruction correspond à la notification de l'état d'avancement du `BackgroundWorker`. Cet avancement correspond à un pourcentage. Grâce à un produit en croix avec la limite haute, on obtient le pourcentage d'avancement. Cette méthode `ReportProgress` lance l'événement `ProgressChange`. Il vous faut maintenant définir comment va être traité cet événement. Pour cela, de la même manière que pour `DoWork`, double-cliquez sur l'événement `ProgressChange` dans la fenêtre de Propriétés. Visual Studio va créer le squelette de la méthode dans laquelle vous pourrez mettre le traitement. À chaque nouvelle étape d'avancement, nous marquerons l'état d'avancement en cours, et nous ferons avancer la `ProgressBar`.

```
Public Sub BackgroundWorker1_ProgressChanged(ByVal
  <> sender As System.Object, ByVal e As System
  <> .ComponentModel.ProgressChangedEventArgs) Handles
  <> BackgroundWorker1.ProgressChanged
    ProgressBar1.Value = e.ProgressPercentage
    TextBox2.Text = "" & counter.ToString()
End Sub
```

`e.ProgressPercentage` correspond au nombre qui aura été passé en argument dans la méthode `ReportProgress`. C'est comme ceci que vous le retrouverez, ce qui permet de faire le lien entre le traitement principal de votre programme et la notification de l'avancement de celui-ci.

Enfin, il vous reste à définir l'action à faire en fin de traitement. Une fois encore, un événement correspond à cela : `RunWorkerCompleted`. Comme pour `DoWork` et `ProgressChange`, double-cliquez sur l'événement correspondant dans la fenêtre de Propriétés et Visual Studio vous créera le squelette de la méthode. On y marquera simplement la fin de traitement dans le `Label` de votre programme.

```
Public Sub BackgroundWorker1_RunWorkerCompleted(ByVal
  <> sender As System.Object, ByVal e As System
  <> .ComponentModel.RunWorkerCompletedEventArgs) Handles
  <> BackgroundWorker1.RunWorkerCompleted
    TextBox2.Text = "Background count finished."
End Sub
```

Maintenant que l'ensemble des événements du `BackgroundWorker` est défini, il ne reste plus qu'à préparer son lancement et à l'affecter au `Click` du bouton. Cliquez sur le bouton, et double-cliquez sur l'événement `Click` dans la fenêtre de Propriétés. Dans cette partie, on créera la limite haute pour le comptage, et on lancera le traitement du

BackgroundWorker après avoir prévenu du lancement du traitement. Voici ce que nous obtiendrons :

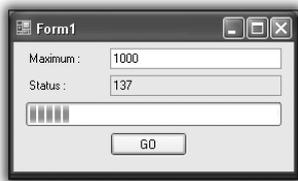
```
Public Sub Button1_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button1.Click
    max = Integer.Parse(TextBox1.Text)
    ProgressBar1.Maximum = 100
    TextBox2.Text = "Launching background count..."
    BackgroundWorker1.RunWorkerAsync ()
End Sub
```



### Accessibilité

Il faudra avoir défini "max" dans une partie du programme accessible par toutes les méthodes.

Réessayons maintenant le lancement du programme avec un grand nombre.



**Figure 9.92 :**  
*Utilisation du BackgroundWorker*

Nous voyons que le programme s'exécute sans bloquer, et nous pouvons suivre l'avancement de notre boucle, jusqu'à sa fin.

Le `BackgroundWorker` est réellement un objet simple d'utilisation qui permet de profiter des bienfaits du multithreading pour exécuter une tâche en arrière-plan. Si une partie de votre programme doit prendre un peu de temps, il est conseillé d'utiliser cet objet plutôt qu'essayer de le gérer à travers des threads, surtout si vous devez agir sur une interface graphique. En effet, avec le framework .Net, la gestion des interfaces graphiques est un peu particulière dans un contexte multithreading. C'est ce que nous allons voir dans la partie qui suit.

## Comment agir sur l'interface utilisateur ?

Merci au `BackgroundWorker`, nous avons pu dans la partie précédente effectuer une action en tâche de fond, notifier notre progression à

l'utilisateur, faire un traitement de fin, et tout cela sans bloquer l'affichage et sans perdre d'interactivité sur notre application.

Supposons maintenant que ce `BackgroundWorker` n'existe pas, et essayons d'utiliser les autres connaissances apprises dans le chapitre.

Vous vous en doutez bien, nous allons utiliser des threads pour effectuer le même traitement.

## Utilisation des threads

Nous allons utiliser la même application, mais nous allons supprimer tout le code relatif au `BackgroundWorker`. Il ne reste plus grand-chose, me direz-vous, c'est pourquoi nous allons remplacer ce code par une gestion de threads créée par nos soins.



*Reportez-vous à ce sujet au début de ce chapitre.*

Comme nous l'avons vu, il faut créer à notre thread un point d'entrée, une méthode qui marquera le début de son action. Celle-ci va effectuer notre action de comptage et retransmettre à l'interface graphique les différentes actions effectuées. Nous l'appellerons `count` :

```
Public Sub Count()  
    Dim count As Integer = Integer.Parse(TextBox1  
    & .Text)  
    Dim i As Integer  
    ProgressBar1.Maximum = count  
  
    TextBox2.Text = "Launching count..."  
  
    For i As Integer = 1 To count  
        ProgressBar1.Value = i  
        TextBox2.Text = "" & i.ToString()  
    Next i  
  
    TextBox2.Text = "Count finished."  
End Sub
```

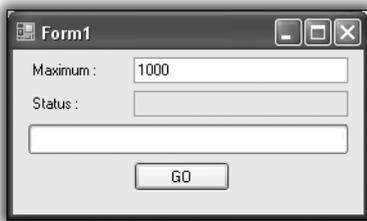
Très bien, maintenant que notre méthode de traitement est créée, il nous faut instancier un thread et lui spécifier que son point d'entrée sera la méthode `count` :

```
Dim thread1 As New Threading.Thread(AddressOf Count)
    thread1.Name = "ThreadCount"
    thread1.Start()
```

Parfait, il n'y a désormais plus qu'à lier cette action au `Click` du bouton de notre interface :

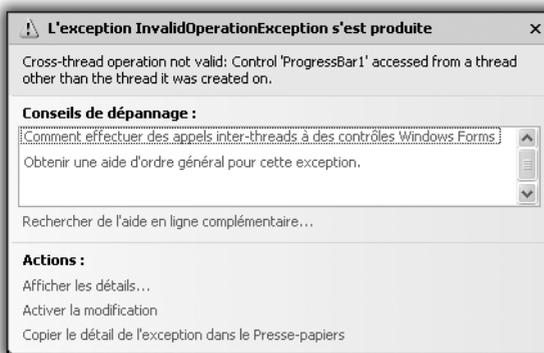
```
Public Sub Button1_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim thread1 As New Threading.Thread(AddressOf Count)
    thread1.Name = "ThreadCount"
    thread1.Start()
End Sub
```

Notre transformation est maintenant prête, notre application va effectuer exactement la même tâche qu'avec le `BackgroundWorker`, mais en utilisant directement un thread. On exécute :



**Figure 9.93 :**  
*Exécution utilisant un thread secondaire*

Et là, contre toute attente :



**Figure 9.94 :**  
*Crash de l'application*

Notre application se termine avec une erreur *Cross Thread Exception*.

## L'exception "Cross thread"

Voici la particularité dont je vous parlais concernant .Net et la gestion des interfaces graphiques dans un contexte multithread.

Le framework .Net met en place une protection qui empêche un thread de modifier une interface graphique si celle-ci n'a pas été construite dans le thread qui appelle la modification.



DEFINITION

### Éléments d'interface graphique

Par élément d'interface graphique, on entend tout objet dérivant de la classe `Control`, tels les `Form`, `Label`, `Button`, `ProgressBar`, `TextBox`...

Dans notre application, le processus principal crée la fenêtre.

D'un autre côté, nous créons pour faire notre traitement un autre thread qui exécutera la méthode `Count`. Or, dans cette méthode, nous essayons de modifier la `ProgressBar` :

```
ProgressBar1.Maximum = count
```

Cependant, nous avons vu qu'elle avait été créée dans le processus principal. C'est lors de cet accès que le framework se protège et envoie cette fameuse erreur *Cross thread exception*.

À ce point-là, nous sommes un peu coincés... On ne sait pas dire "*Thread 1, fais cette instruction, puis Thread 2, fais cette instruction, et revenons à Thread 2*". Car ce serait la solution. Nous effectuons le traitement non graphique dans notre thread secondaire, puis au moment de traiter les éléments graphiques on repasse dans le thread d'origine.

C'est à peu près la solution, sauf qu'on ne peut pas traiter le passage interthread par instructions. Il faudra le faire à partir de méthodes, en utilisant des délégués.

## La solution : les délégués et la méthode `Invoke`

La solution est de faire exécuter par un `Control` toute méthode qui le modifie dans son thread d'origine.

Par exemple, nous avons vu que `count` modifie la `ProgressBar`. Donc, si le thread qui appelle `count` n'est pas celui qui a créé la `ProgressBar`, il faudra faire l'échange. Le thread qui a créé la `ProgressBar` va alors exécuter `count`.

La méthode qui permet de faire cela est la méthode `Invoke`. C'est une méthode qui existe pour tous les objets de type `Control` et qui va renvoyer l'exécution d'une autre méthode au thread d'origine.

Pour pouvoir utiliser `Invoke`, il faut en revanche créer un délégué correspondant à la méthode à *Invoker*.



*Reportez-vous à ce sujet au chapitre [Dialoguez avec un ordinateur](#).*

```
Public Delegate Sub DelegateCounter()
```

Nous avons créé le délégué qui correspond à notre méthode `count`.

Maintenant, comme nous savons que notre méthode `count` va modifier la `ProgressBar`, au moment d'appeler `count` il nous faudra, plutôt que le faire directement, utiliser `Invoke`. `Invoke` prend comme paramètre la méthode à *Invoker*.

```
Dim myDelegate As New DelegateCounter(AddressOf Count)
ProgressBar1.Invoke(myDelegate)
```

De cette manière, l'accès au `Control` est sécurisé. Vous pouvez maintenant exécuter de nouveau l'application, qui pourra se poursuivre jusqu'à la fin.



### Count

La méthode `count` modifie également d'autres `Control`, comme la `TextBox` ou le `Label`. Or ces éléments ont été créés dans le même thread que la `ProgressBar`, donc l'`Invoke` de celle-ci fonctionne également pour eux.



**Figure 9.95 :**  
*Exécution utilisant l'Invoke*

Une bonne pratique concernant l'Invoke est l'utilisation d'une méthode `Invoker` qui va faire la vérification de la nécessité de l'Invoke. En effet, les objets de type `Control` possèdent également une propriété `InvokeRequired` qui permet de savoir à l'exécution s'il est nécessaire de faire un `Invoke` ou non.

De cette manière si l'Invoke est nécessaire, vous le faites, sinon vous appelez la méthode normalement. Il est recommandé de faire ceci car faire un `Invoke` est gourmand en ressources et peut réduire vos performances. L'utilisation de la propriété `InvokeRequired` permet d'éviter de le faire systématiquement. En définissant cette vérification dans une autre méthode, vous optimisez vos appels, et il suffit ensuite d'appeler l'Invoker pour garder la sécurité tout en limitant les pertes de performances.

```
Public Sub Invoker()  
    Dim myDelegate As New DelegateCounter(AddressOf  
    < Count)  
    If (ProgressBar1.InvokeRequired) Then  
        ProgressBar1.Invoke(myDelegate)  
    Else  
        Count()  
    End If  
End Sub
```

## Bien gérer les erreurs avec plusieurs processus

Nous allons maintenant aborder un point délicat (un de plus) dans la gestion multithread : la gestion des erreurs. Dans notre application, nous nous sommes arrangés pour qu'il n'y ait pas d'erreur, mais ce ne sera pas toujours le cas.

Réutilisons notre application dans le cas où elle crée une erreur *Cross Thread Exception*, par exemple.

```
Public Sub Button1_Click(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles Button1.Click  
    Dim thread1 As New Threading.Thread(AddressOf Count)  
    thread1.Name = "ThreadCount"  
    thread1.Start()  
End Sub
```

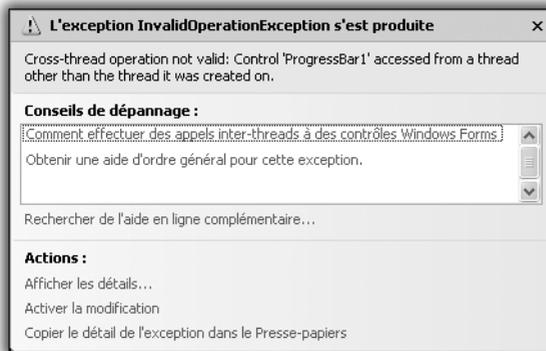
Maintenant, tentons d'éviter l'erreur en utilisant un bloc `Try Catch`.



*Reportez-vous à ce sujet au chapitre **Rendre un programme robuste**.*

```
Try
Dim thread1 As New Threading.Thread(AddressOf Count)
thread1.Name = "ThreadCount"
thread1.Start() 'démarrage la 1ère opération dans un thread
Catch ex As Exception
Console.WriteLine("il y a eu une erreur dans le thread
                    secondaire")
End Try
```

L'utilisation d'un bloc `Try Catch` étant censée rattraper les erreurs, au pire, notre méthode `count` ne s'exécutera pas comme il faut, mais au moins elle ne plantera pas l'application.



**Figure 9.96 :**  
*Exécution protégée  
par un Try Catch*

Perdu, l'application a quand même planté avec une erreur *Cross Thread Exception*. Pourquoi ?



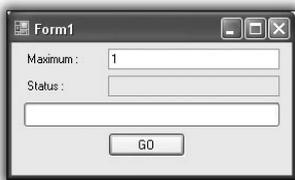
### Try et Catch

Un bloc `Try Catch` protège des exceptions survenant dans le thread qui a exécuté ce bloc `Try Catch`.

Dans notre cas, nous avons un bloc `Try Catch` qui fait appel à un autre thread. Il n'y a eu aucun problème dans notre thread principal. C'est notre thread secondaire, qui, en voulant modifier un élément graphique qu'il n'avait pas créé, a provoqué une erreur.

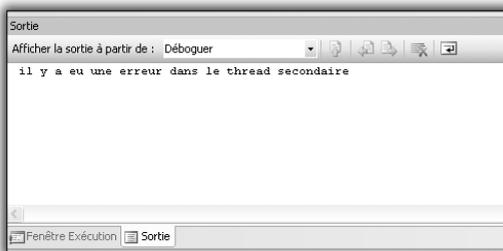
Très bien, protégeons donc notre second thread en mettant un bloc Try Catch dans notre méthode `count`, et exécutons de nouveau :

```
Public Sub Count()  
    Try  
        Dim count As Integer = Integer.Parse(TextBox1  
        &< .Text)  
        ProgressBar1.Maximum = count  
  
        TextBox2.Text = "Launching count..."  
  
        For i As Integer = 1 To count  
            ProgressBar1.Value = i  
            TextBox2.Text = "" & i.ToString()  
        Next i  
  
        TextBox2.Text = "Count finished."  
    Catch ex As Exception  
        Console.WriteLine("il y a eu une erreur  
        dans le thread secondaire")  
  
    End Try  
End Sub
```



**Figure 9.97 :**  
*Thread secondaire protégé par un bloc Try Catch*

Il ne s'est rien passé. Rien de visible, en tout cas. De plus, Visual Studio nous a bien fait état d'une erreur *Cross Thread Exception*. Cependant, l'application n'a pas planté, elle est toujours active, ce qui n'était pas le cas précédemment. Voyons la fenêtre de sortie.



**Figure 9.98 :**  
*Sortie du thread secondaire protégé*

Nous sommes passés dans la partie `Catch`. C'est une bonne nouvelle, car cela nous montre bien que le thread secondaire a été protégé. Ceci nous amène à une conclusion importante.



REMARQUE

**Protection sur les erreurs**

Tous les threads de votre programme doivent gérer la protection de leurs erreurs.

Comme nous avons pu le constater tout au long de ce chapitre, la gestion d'un contexte multithread est une chose très subtile. Cependant, elle donne à vos applications une nouvelle dimension, car elle permet de bénéficier d'une activité accrue, ainsi que profiter du maximum de ressources présentes sur le système. Cela demande en contrepartie une rigueur et une attention particulières. Ne vous inquiétez pas si cela vous pose des soucis, car même des professionnels se trouvent régulièrement confrontés à des erreurs qu'ils ne comprennent pas ou qu'ils ont du mal à solutionner à cause des threads. Cela demande également de la pratique et de la curiosité. N'hésitez pas à jouer avec les threads, à comparer, à en ralentir exprès avec la commande `Sleep` pour voir le comportement, mais tentez aussi de diviser vos applications en sous-traitements exécutables par plusieurs threads. Elles prendront tout de suite une autre dimension, croyez-moi. Bon courage !