Les threads en C#

Table des matières

Cet article permet d'apprendre à créer et à utiliser des threads managés par le framework .NET avec le langage C#. L'article explique également comment passer des paramètres a un thread et une façon simple de protéger une zone de code critique.

N'hésitez pas à commenter cet article ! Commentez ⭐⭐⭐⭐⭐

Article lu 66758 fois.

L'auteur

Olivier Brin

L'article

Publié le 10 janvier 2005

Version PDF Version hors-ligne

ePub, Azw et Mobi

Liens sociaux

Remerciements▲

Je tiens à remercier **neo.51** et **abelman** pour l'accueil au sein de l'équipe .NET de developpez.com ainsi que **Ukyuu** et **Pascal Jankowski** pour la relecture et la correction du présent article.

I. Le multithreading▲

De tout temps, l'homme a cherché à améliorer sa productivité en parallélisant ses tâches. Par exemple, les principes du montage à la chaîne ou des architectures pipelinées découlent directement de cette envie d'optimisation. En programmation aussi, il est possible de réaliser ce genre de mécanisme.

Pour illustrer ce besoin, prenons un exemple simple : un programme effectue divers calculs compliqués pendant un temps relativement long. L'utilisateur souhaite voir les résultats intermédiaires apparaître sur son interface graphique en temps réel. On peut clairement séparer dans ce cas la partie calcul et l'autre partie concernant l'affichage. Plutôt que de parler de parties, parlons de tâches.

Si nous exécutons le programme de manière séquentielle, soit sans parallélisme, il y a de fortes chances pour que l'utilisateur doive attendre la fin des calculs pour qu'un affichage des résultats apparaisse enfin. En séparant les tâches calculs et affichage, chacune aura, à tour de rôle un petit moment alloué par le processeur pour travailler. Cette méthode de programmation est aujourd'hui largement utilisée et contribue beaucoup à améliorer la réactivité des applications.

Il faut noter que sur un système monoprocesseur, la notion de multithreading est toute relative. En effet, le processeur ne peut réellement travailler que dans une tâche à la fois. Cela dit, le changement de contexte entre deux tâches se faisant à une vitesse si rapide, qu'il nous semble que le travail se fait simultanément et que les deux tâches travaillent en même temps. Avec un système biprocesseur par contre, si nous avons deux tâches, elles seront dispatchées sur un des processeurs disponibles et elles travailleront réellement en parallèle.

II. La classe System.Threading.Thread▲

En programmation pure, on ne parle plus de tâches, mais de **threads** ou de **processus.** Il existe une différence entre ces définitions mais l'expliquer nous conduirait à nous éloigner de l'objectif de ce cours. Par la suite, je parlerai donc de threads.

En C# et dans tous les autres langages du framework .NET, la classe Thread se trouve dans l'espace de noms System.Threading. Pour simplifier, un objet de la classe Thread symbolise une tâche. L'utilisation des threads avec .NET a été fortement simplifiée comparée aux méthodes natives Win32. Nous allons dans ce cours voir comment créer, utiliser et détruire des threads managés, ainsi que quelques mécanismes de protection de ressources critiques.

II-A. Créer et lancer un thread▲

Le code suivant montre comment créer et lancer un thread managé :

Code complet de création d'un thread C#
Sélectionnez

```
using System;
using System.Threading;

class ThreadedApp
{
    public static void Main()
    {
        // Déclaration du thread
        Thread myThread;

        // Instanciation du thread, on spécifie dans le
        // délégué ThreadStart le nom de la méthode qui
        // sera exécutée lorsque l'on appele la méthode
        // Start() de notre thread.
        myThread = new Thread(new ThreadStart(ThreadLoop));

        // Lancement du thread
        myThread.Start();
    }

    // Cette méthode est appelé lors du lancement du thread
    // C'est ici qu'il faudra faire notre travail.
    public static void ThreadLoop()
    {
        // Tant que le thread n'est pas tué, on travaille
            while (Thread.CurrentThread.IsAlive)
            {
            // Attente de 500 ms
            Thread.Sleep(500);

            // Affichage dans la console
            Console.WriteLine("Je travaille...");
        }
    }
}
```

Cet exemple crée un thread managé qui affiche, toutes les 500 millisecondes, le texte « Je travaille » dans la console système. Nous allons décortiquer les points clés de ce programme.

Sélectionnez

```
// Déclaration du thread
Thread myThread;
```

Comme pour tous les autres types, un thread doit être déclaré avant d'être instancié. On pourrait également utiliser le nom complet de la classe Thread, soit System.Threading.Thread et ainsi éviter la directive using System.Threading. Le nom de notre thread sera donc **myThread.**

Sélectionnez

```
// Instanciation du thread, on spécifie dans le
// délégué ThreadStart le nom de la méthode qui
// sera exécutée lorsque l'on appele la méthode
// Start() de notre thread.
myThread = new Thread(new ThreadStart(ThreadLoop));
```

C'est ici que l'on crée notre objet qui va représenter notre tâche. Le constructeur de la classe Thread introduit un concept spécifique de .NET, les délégués. Pour résumer, considérons un délégué comme un pointeur de méthode. Le délégué ThreadStart prend comme argument le nom d'une méthode que le thread va exécuter lorsqu'il sera lancé. La méthode passée en argument sera appelée lorsque l'on invoquera la méthode Start de notre thread. C'est dans cette méthode **ThreadLoop** que le travail que l'on veut paralléliser devra être introduit.

Sélectionnez

```
// Lancement du thread
myThread.Start();
```

La méthode **Start** permet de lancer le thread, soit implicitement, d'exécuter la méthode déléguée par ThreadStart.

Sélectionnez

```
// Tant que le thread n'est pas tué, on travaille
while (Thread.CurrentThread.IsAlive)
```

Nous sommes désormais dans notre méthode **ThreadLoop.** Généralement, on trouve une boucle qui tournera tant que le thread est en vie. Pour faire ce contrôle, on peut utiliser le membre **IsAlive** qui renvoie **vrai** si le thread est toujours en exécution. Le membre statique Thread.CurrentThread renvoie lui la référence sur le thread actuellement en exécution. Ainsi, tant que nous ne tuons pas le thread par des moyens explicites, nous allons rester dans cette boucle.

Sélectionnez

```
// Attente de 500 ms
Thread.Sleep(500);

// Affichage dans la console
Console.WriteLine("Je travaille...");
```

Le code précédent est exécuté à chaque passage dans la boucle. On utilise la méthode statique **Sleep** de la classe Thread pour ordonner à notre thread de passer en attente passive durant 500 ms. Un fois ce délai écoulé, on affiche un texte dans la console.

II-B. Passer des paramètres a un thread▲

La méthode de création de threads avec le délégué ThreadStart ne permet pas de passer directement des paramètres. Il existe un moyen très simple pour contourner ce souci : il suffit de créer une classe spécifique qui contient la méthode du thread (**ThreadLoop** dans l'exemple ci-dessus). On utilisera alors les membres de cette classe comme paramètres.

La classe suivante a le rôle de gérer le thread. On trouve dans cette classe la définition de la méthode utilisée par le thread ainsi que des champs membres utilisés comme paramètres. Il suffira alors de modifier ces champs avant de créer notre thread pour les utiliser par la suite dans notre méthode **ThreadLoop.** On peut également modifier ces champs membres durant l'exécution du thread pour changer le comportement de notre méthode ThreadLoop.

Dans l'exemple suivant, le type du paramètre est un entier. Il est bien sûr possible d'utiliser tous les types proposés par le framework .NET comme paramètre.

Classe de gestion de thread
Sélectionnez

```
public class MyThreadHandle
{
    // Cet entier sera utilisé comme paramètre
    int myParam;

    // Constructeur
    public MyThreadHandle (int myParam)
    {
        this.myParam = myParam;
    }

    // Méthode de modification du paramètre
    public void SetParam(int param)
    {
        this.myParam = param;
    }

    // Méthode boucle du thread
    public void ThreadLoop()
    {
        // On peut utiliser ici notre paramètre myParam
        switch (myParam)
        {
            // ...
        }
    }
}
```

Dans l'exemple ci-dessus, le membre **myParam** sera utilisé comme paramètre du thread. C'est-à-dire que, via l'accesseur **SetParam** ou le constructeur de la classe, il est possible de le modifier. La subtilité réside dans le fait que la méthode du thread ThreadLoop est membre de la même classe que myParam. Ainsi, notre méthode peut accéder aux différents champs membres que l'on peut créer et modifier à sa guise.

La création du thread est quelque peu modifiée avec cette technique. Voyons comment créer le thread avec un paramètre :

Exemple de création d'un thread avec paramètre
Sélectionnez

```
// On crée notre 'manipulateur' de thread en y passant un
// paramètre classique
MyThreadHandle threadHandle = new MyThreadHandle(10);

// On crée notre thread en y donnant comme méthode boucle, une
// méthode membre de notre manipulateur
Thread t = new Thread(new ThreadStart(threadHandle.ThreadLoop));

// La méthode ThreadLoop de l'objet threadHandle est appelée, et myParam est donc acces
t.Start();
```

En premier lieu, il faut créer une instance de la classe **MyThreadHandle**. Le constructeur de cette classe prend en paramètre un entier, qui sera attribué au membre **myParam**. Ensuite, on crée le thread en déléguant la méthode publique **ThreadLoop**, membre de la classe **MyThreadHandle** et appartenant à notre objet **threadHandle**.

En appelant la méthode **Start** de notre thread, la méthode **ThreadLoop** de l'objet **threadHandle** sera exécutée et aura accès au membre **myParam.** Ainsi, nous avons paramétré l'exécution de notre thread. Il est bien sûr possible de mettre en place toutes sortes de membres qui seront utilisés par la méthode boucle du thread, comme des références sur d'autres objets. Ce mécanisme est très simple mais permet de régler le problème du passage de paramètres à une méthode déléguée et ceci de manière élégante et logique.

II-C. Protéger des ressources critiques▲

Une ressource est dite critique si sa modification ne doit pas être interrompue. Par exemple, considérons une zone de code critique de 5 lignes. Si la tâche se trouve actuellement à la 3e ligne de cette zone et qu'elle perd le processeur, certaines données seront erronées, ou perdues, voire pire encore. Il faut donc préciser au CLR qu'une certaine zone de code ne doit pas être interrompue par d'autres tâches.

Pour réaliser ceci, C# propose un mécanisme extrêmement simple avec le mot-clé **lock**. La directive lock avant une portion de code ou une déclaration permettra d'éviter le problème de l'exclusion mutuelle (mutex). Imaginons que plusieurs tâches accèdent à une ressource critique pour y débiter un montant. La ressource critique se nomme Solde :

Exemple de protection d'une zone de code critique simple
Sélectionnez

```
private void DebiterCompte(int Montant)
{
    // Le code dans le bloc suivant sera protégé
    lock (this)
    {
        Console.WriteLine("Solde avant transaction : " + Solde);
        Console.WriteLine("Montant à débiter : " + Montant);

        // Code critique
        Solde = Solde - Montant;

        Console.WriteLine("Solde après transaction : " + Solde);
    }
}
```

Lorsqu'une tâche voudra débiter le compte, elle appellera la méthode **DebiterCompte** en lui passant un montant, et si aucune autre tâche n'est actuellement en train d'exécuter la partie protégée, elle pourra entrer dans la zone de code critique. Sinon, la tâche sera placée dans une file d'attente.

Il existe d'autre mécanismes de protection de ressources critiques, comme les classes Monitor (**lock** est en fait un raccourci qui utilise la classe Monitor), la classe Mutex. Celle-ci permet de créer vos propres mécanismes de synchronisation.

II-D. Stopper des threads▲

A tout moment, on peut être amené à vouloir explicitement détruire des threads, par exemple lors de la fermeture du programme, histoire de faire les choses proprement. La méthode la plus sèche pour stopper un thread se nomme **Abort.** Celle-ci tue le thread et lève une exception du type ThreadAbortException.

Sélectionnez

```
// Détruit notre thread
myThread.Abort();
```

Une autre méthode utile, mais dont l'effet est différent, est **Suspend.** En appelant cette méthode, le thread sera mis en attente jusqu'au moment où la méthode **Resume** sera appelée sur le thread en question.

Sélectionnez

```
// Suspend le thread
myThread.Suspend();

// Dans cette zone, le thread ne tourne plus
// ...

// Le thread reprend son activité
myThread.Resume();
```

Vous avez aimé ce tutoriel ? Alors partagez-le en cliquant sur les boutons suivants :

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteur. Copyright © 2005 Olivier Brin. Aucune reproduction, même partielle, ne peut être faite de ce site ni de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Envie de quitter Paris pour partir travailler dans l'IT en région ? LAOU, une plateforme de recrutement,

Inscrivez-vous gratuitement à l'édition 2020 du .NET Challenge France organisé par SoftFluent

ReSharper Ultimate 2020.1 est disponible avec la prise en charge améliorée de C# 8.0 et C++ 20

organise un webinaire le 18 juin prochain

Mise en place d'un service d'authentification avec le framework IdentityServer4, un tutoriel de Hinault Romaric

Développeur Java Client Final Ecologie
↳ EASY PARTNER - Ile de France - Paris (75009)

Technicien(ne) de développement électronique
↳ KELENN TECHNOLOGY - Ile de France - Igny (91430)

Ananalsyte-technicien D'exploitation Informatique H/f - Cdd 6 Mois
↳ CAISS RETR PREV CLERCS EMPLOYES NOTAIRES - Ile de Fran Voir plus d'offres

Responsable bénévole de la rubrique Microsoft DotNET : Hinault Romaric - Contacter par email

Nous contacter Participez Hébergement Informations légales Partenaire : H bergement Web

© 2000-2020 - www.developpez.com