# Threads et c#

## I) Créer et lancer un thread

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.

```
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...");
                }
        }
}
```

- **Créer un projet console en C# et tester ce code.**

## II) Passer des paramètres à 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 attributs 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 attributs utilisés comme paramètres. Il suffira alors de modifier ces attributs avant de créer notre thread pour les utiliser par la suite dans notre méthode **ThreadLoop()**. On peut également modifier ces attributs durant l'exécution du thread pour changer le comportement de notre méthode **ThreadLoop().**

```
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, l'attribut **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 :

```
// 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 accessible!
t.Start();
```

En premier lieu, il faut créer un objet instance de la classe **MyThreadHandle**. Le constructeur de cette classe prend en paramètre un entier, qui sera attribué au membre **myParam**, ici 10.
Ensuite, on crée le thread en utilisant la méthode publique **ThreadLoop()** de l'objet **threadHandle**, (**ThreadLoop()** est membre la classe **MyThreadHandle**).
En appelant la méthode Start de notre thread, la méthode **ThreadLoop()**de l'objet **threadHandle** sera exécutée et aura accès à l'attribut **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.

- **Créer un projet console en C# et tester ce code.**

## III) 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 3ème 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**. Imaginons que plusieurs tâches accèdent à une ressource critique pour y débiter un montant. La ressource critique se nomme *Solde* :

```
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);
        }
}
```

## IV) Stopper un thread

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.**

```
// 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.

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

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

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

## V) TP

Une application (C# Windows form) doit extraire d'un fichier journal les données relatives au fonctionnement d'une station de relevage. La station est répartie sur deux sites comprenant chacun deux pompes (voir JournalPompes.txt).
Deux threads de l'application accèdent au fichier. Un thread extrait les données (ligne/ligne) relatives au site A et les affiche. Un autre thread extrait les données (ligne/ligne) relatives au site B et les affiche.
Le thread 1 ajoute OKA à la fin de chaque ligne du fichier relative au site A, le thread 2 ajoute OKB à la fin de chaque ligne du fichier relative au site B.