Introduzione al multithreading

Il Multithreading consiste nella possibilità di scomporre le attività di elaborazione di un programma applicativo in più parti, dette Thread, in modo che

- esse possano essere eseguite in parallelo da processori diversi,
- l'avanzamento di una attività non sia vincolato al completamento di un'altra.

Si possono avere **Thread indipendenti** che svolgono attività completamente disgiunte e parallele (es.: il download di due diversi file a cura di un browser), oppure **Thread che collaborano** al raggiungimento di un risultato (es.: suddivisione di un gran numero di calcoli da fare tra più thread che poi restituiscono ciascuno un risultato parziale da far confluire nel risultato finale), oppure **Thread che competono** per l'uso di una risorsa condivisa (es.: due attività di prenotazione dei posti in un volo aereo che vanno ad aggiornare l'elenco dei posti a disposizione)

In generale, la programmazione dei Thread è complicata dalle problematiche di sincronizzazione degli stessi e dal fatto che i tempi di esecuzione delle attività non è mai prevedibile a priori, perché dipende dagli algoritmi di scheduling del sistema operativo e da eventi esterni quali le azioni di input/output di un utente.

Si dice anche, in generale, che le attività svolte dai Thread sono "asincrone", ovvero ciascuna avanza con i suoi tempi, senza preoccuparsi dei tempi di avanzamento degli altri Thread.

Dalla versione 5 di C# è stata introdotta la libreria TPL (Task Parallel Library) che semplifica molto la programmazione dei Thread spostando l'attenzione agli aspetti logici delle attività che essi devono svolgere (Task) senza occuparsi delle questioni legate alle assegnazioni dei processori o dei dettagli di implementazione dei meccanismi di sincronizzazione degli stessi.

La classe Task (in System.Threading.Tasks) consente di programmare facilmente le attività da far svolgere ai Thread automaticamente gestiti dal sistema operativo.

Si possono avere Task con o senza parametri in input e che restituiscono o meno un risultato.

Per creare un Task si utilizza il costruttore che vuole un "delegato", ovvero un riferimento ad un metodo oppure una espressione lambda.

Il primo esempio consiste nella creazione di un Task che stampa un messaggio su Console:

```
using System;
using System.Threading.Tasks;

public static void Main()
{
  // Create a task and supply a user delegate by using a lambda expression.
  Task t = new Task( () => Console.WriteLine("Hello from the task"));
  // Start the task.
  t.Start();
  // Output a message from the calling thread.
```

```
  for(int i=0; i<100000; i++) { /*  ciclo di passatempo  */}
  Console.WriteLine("Hello from main thread”);
  t.Wait();
}
```

Si può ottenere la stampa di

Hello from the task

Hello from main thread

oppure di

Hello from main thread

Hello from the task

senza che sia prevedibile a priori l’ordine di esecuzione delle attività.

L’istruzione t.Wait() fa in modo che il Main() prima di terminare aspetti il completamento dei lavori del Task t.

Il secondo esempio propone due Task indipendenti -- con passaggio di parametri in input

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
      public static void Main()
      {
            Console.WriteLine("Hello World");
            int  dato = 1;
            Task t1 = new Task((p)=>{Console.WriteLine("t1 " + p);}, dato);
            dato = 2;
            Task t2 = new Task((p)=>{Console.WriteLine("t2 " + p);}, dato);
            dato = 3;
            Task t3 = new Task(()=>{Console.WriteLine("t3 " + dato);});
            dato = 4;
            Task t4 = new Task(()=>{Console.WriteLine("t4 " + dato);});
            dato = 5;
            t1.Start(); t2.Start(); t3.Start(); t4.Start();
            t1.Wait();
            t2.Wait();
            t3.Wait();
            t4.Wait();
      }
}
```

si ottiene:

Hello World
t1 1
t2 2
t3 5
t4 5

Si noti che i Task t1 e t2 prevedono in input un parametro p a cui viene assegnato il valore della variabile dato. Il loro comportamento è del tutto ovvio e prevedibile.

Invece i Task t3 e t4 vanno ad utilizzare direttamente la variabile dato che è esterna al Task.

Sia t3 che t4 si trovano ad utilizzare l'ultimo valore assegnato alla variabile esterna dato, prima dello Start, che è 5.

Tipicamente, l'espressione che esprime le attività del Task possono fare riferimento ad un oggetto esterno da utilizzare nelle elaborazioni previste dal Task stesso.

Se ho un array di Task di cui aspettare il completamento di tutte, posso usare l'istruzione

```
Task.WaitAll(taskArray);
```

Esempio 3 diTtask paralleli che COOPERANO per svolgere dei calcoli su una array "molto grande"

```
// si usa la classe Task<TipoDelRisultato>

int[] arr = new int[]{1,2,3,4,5,6,7,8,9,10};

Task<int> t5 = new Task<int>(()=> {
            int ris = 0;
            for(int j=0; j<5; j++)
            { ris = ris+arr[j]; }
            return ris;
            });
Task<int> t6 = new Task<int>(()=> {
            int ris = 0;
            for(int j=5; j<10; j++)
            { ris = ris+arr[j]; }
            return ris;
            });
t5.Start(); t6.Start();
// si aspetta che entrambi abbiano il risultato!
int somma = t5.Result + t6.Result;
Console.WriteLine("somma = " + somma);  // 55
}
}
```

La mutua esclusione

Quando due Task devono agire sulla stessa variabile, ovvero sono in COMPETIZIONE, si può creare una situazione di conflitto qualora le modifiche effettuate da un Task si sovrappongano a quelle effettuate dall'altro Task.

Ad esempio due Task potrebbero entrambi leggere "posto libero" e successivamente inserirvi il nome di un diverso cliente, con il risultato di avere la prima registrazione effettuata successivamente sovrascritta dalla seconda.

Occorre pertanto un qualche meccanismo per garantire l'accesso esclusivo ad una variabile da parte dei Task che competono per l'utilizzo della stessa. Si tratta sostanzialmente di applicare il concetto di "semaforo", che risulta familiare a chi viaggia per strada e così può entrare tranquillamente nell'area dell'incrocio di due strade.

L'oggetto **SemaphoreSlim** (in System.Threading) **con valore 1 (si tratta di un counting Semaphore) consente ad un solo Task alla volta di effettuare delle operazioni su variabili condivise.**

```
public class Volo
{
  // variabile condivisa
   private string posto;
  // semaforo
  private SemaphoreSlim ss;

public Volo()
{  posto = "libero";  ss = new SemaphoreSlim(1); }

// metodo di prenotazione
public void Prenota(string nomeCliente)
{
   // aspetto al semaforo finché esso diventa"verde"
   ss.WaitAsync();
   // operazioni sulla variabile condivisa
   if (posto == "libero")
```

```
   {  posto = nomeCliente; }
   ss.Release(); }
}
public string LeggiPosto()
{  return posto; }
}

class Program
{
public static void Main()
{
Volo v = new Volo();
Task t7 = new Task(()=>{v.Prenota("Antonio");});
Task t8 = new Task(()=>{v.Prenota("Gianni");});
t7.Start(); t8.Start();
t7.Wait(); t8.Wait();
Console.WriteLine(v.LeggiPosto());
}
```

Alla fine uno dei due clienti avrà prenotato e l'altro no. Chissà a chi spetterà il posto!!!

Chiamare il metodo WaitAsync sul semaforo produce una attesa che terminerà quando il Task riesce ad accedere al semaforo.

Un altro esempio di COMPETIZIONE sull'uso di una risorsa.

L'implementazione della classe Scuola sfrutta il parallelismo dei Task (lo incapsula e dall'esterno il chiamante non ne sa nulla!)

```
public class Scuola
```

```
{
      private int[] arrVoti;
      private int nVoti;
      private SemaphoreSlim ss; // per controllare l'accesso alla risorsa
                                   condivisa
      private int sommaparziale;  // risorsa condivisa oggetto di competizione


      public Scuola()
      {
            ss = new SemaphoreSlim(1);
            nVoti = 10;
            arrVoti = new int[]{1,1,1,1,1,1,1,1,1,1};
            sommaparziale = 0;
      }

      public void CalcolaSomma()  // con COMPETIZIONE
      {
         Task t8 = new Task( ()=>{ ss.WaitAsync();
                                   sommaparziale = sommaparziale + 4;
                                   ss.Release(); });
         Task t9 = new Task( ()=>{ ss.WaitAsync();
                                   sommaparziale = sommaparziale + 6;
                                   ss.Release(); });
         t8.Start(); t9.Start();
         t8.Wait(); t9.Wait();
      }

      public double CalcolaMedia()
      {
          CalcolaSomma();
          return (double)sommaparziale/nVoti;
      }
}



class Program

{

public static void Main()

{

      Scuola s = new Scuola();

      Console.WriteLine("media " + s.CalcolaMedia());
```

```
}
}
```

SI OTTIENE media 1

perché è importante aspettare t8 e t9?

Se non aspetto che cosa si ottiene???

risposta:

altrimenti si otterrebbe erroneamente

media 0

========== delegati ==========

La creazione di un Task richiede un delegato oppure una espressione lambda.

Ecco 3 diversi modi per creare un Task:

```
public static void ScriviT()
{  Console.WriteLine("t1 ");  }

public static void Main()
{
      Task t3 = new Task( new Action(ScriviT) );
      Task t3 = new Task( () => { ScriviT(); } );
      Task t3 = new Task( delegate { ScriviT(); } );
}
```

A mio parere la sintassi che utilizza una espressione lambda risulta essere la più chiara e leggibile!