Multithreading in ambiente C#

http://escher07.altervista.org

Generalità

Supponiamo di avere una procedura che deve eseguire un serie di elaborazioni simili su argomenti diversi, come ad esempio l'aggiornamento di una serie di tabelle di un database con dati prelevati da un altro e/o ricavati in vario modo.

Il termine serie è abbastanza improprio nel senso che i tasks non devono essere eseguiti necessariamente in modo sequenziale : il loro ouput deve essere però in qualche modo sincronizzato nel senso che ciascuno di questi processi produce una serie di righe di log che per non rendere il tutto incomprensibile non possono mischiarsi.

Per affrontare questo problema inizialmente ho adottato la soluzione classica di chiamate in sequenza della stessa routine, diciamo, per capirsi, qualcosa di questo tipo:

```
UpdateTable ("Tabella_1");
WriteLog ();
UpdateTable ("Tabella_2");
WriteLog ();
...
UpdateTable ("Tabella_N");
WriteLog ();
```

Il risultato alla fine è arrivato, ma due erano di sicuro i punti non completamente soddisfacenti:

- Congelamento della form grafica per tutta la durata dell'elaborazione
- Tempi di elaborazione decisamente lunghi

Mi sono chiesto pertanto se questa non fosse l'occasione per provare qualcosa di un argomento per me totalmente nuovo : la programmazione multithreading.

Concetti

Un thread è in pratica una sessione di elaborazione associata ad un task, ovvero un compito da eseguire, ovvero ancora ad una routine.

I thread possono correre uno "parallelamente" all'altro, ma fra le altre sono importanti due condizioni:

- che i thread non si "sporchino" i dati comuni a vicenda (condizione detta di thread safe)
- che i thread siano in qualche modo sincronizzabili, alla fine dell'elaborazione (un pò come una serie di file di sportelli a cui le persone, dopo essersi sparpagliate si ridispongono in sequenza ordinata)

La questione è in realtà molto meno banale, ma per gli scopi della presente trattazione va bene anche fermarci qui, rimandando per gli eventuali approfondimenti alla documentazione in rete.

Implementazione base in C#

L'ambiente .NET richiede per la programmazione multithreading che, essenzialmente, il metodo o i metodi operativi siano incapsulati in una classe.

Questa classe avrà le sue proprietà (inizializzate attraverso il suo costruttore) come tutte : le proprietà conterranno i parametri da passare alla routine di elaborazione.

Avrà poi una void di esecuzione - che qui si è scelto di chiamare Exec() - che sfruttando dette proprietà si occuperà del passaggio dei dati alla routine di cui sopra.

La particolarità più importate è la presenza di una proprietà privata di tipo callback. Che cos'è il tipo callback : è un tipo dichiarato all'esterno della classe come delegato.

La Exec(), una volta eseguita l'elaborazione chiamerà questa callback passandogli gli argomenti di ritorno (in questo caso la stringa di log dell'elaborazione).

Per capirsi quindi avremo:

```
private SyncRoutineCallback objCallback;

class SyncRoutine
{
      string strTableName;

      private SyncRoutineCallback objCallback;

      private void UpdateTable(string strTable, ref string strLogMessages)
      {
      …
      }


      public void Exec()
      {

      string strMessSyncro ="";
      UpdateTable (strTableName, ref strMessSyncro);

      if (objCallback != null)
         objCallback(strMessSyncro);

      }
}
```

Questo esempio l'ho ottenuto semplificando il programma reale e non ho testato se funziona quindi potrebbero esserci errori banali : la sostanza però è questa!

Veniamo ora al main, ovvero a chi utilizzerà quanto fatto finora. Qui verrà istanziata la nostra classe con tutte le sue proprietà fra cui la callback. Questa conterrà il puntatore ad una opportuna funzione CallBackResult dichiarata nel main e fatta per operare sui valori di ritorno. In pratica avremo quanto segue:

```
SyncRoutine objSR1 = new SyncRoutine("ARTICOLI", new
SyncRoutineCallback(CallbackResult));
```

dove:

```
public void CallbackResult(string RetMsgSyncro)
{
    //  Fa qualcosa con il RetMsgSyncro, ad esempio lo analizza
    //  o lo mette dentro un textbox della main form
}
```

Il cerchio si chiude con l'apertura del Thread ed il passaggio a questo del puntatore alla Exec dell'istanza corrente della classe sopra definita:

```
Thread t1 = new Thread(new ThreadStart(objSR1.Exec));
t1.Start();
```

A questo punto parte la Exec, fa il suo lavoro e valorizza la `strMessSyncro` *che viene passata alla callback della SyncRoutine e da questa alla CallBackResult della main con ad esempio la visualizzazione del nostro messaggio.*

Tutto bene e la sincronizzazione?

In effetti con questa architettura i thread non sono sincronizzati. Per poterlo fare una soluzione è sostituire la chiamata del thread così:

```
ThreadPool.QueueUserWorkItem(new WaitCallback(InvokeSyncRoutine), objSR1);
```

In pratica invece che chiamare direttamente la Exec della Sync Routine si chiama una routine che a sua volta chiama la Exec:

Il resto funziona sempre allo stesso modo con il ritorno dei dati tramite il delegato e la CallbackResult. Il fatto di aver chiamato i thread non come singoli ma come parti di un TreadPool ci consente ora di adoperare dei metodi aggiuntivi per il nostro scopo.

In pratica abbiamo adesso quanto segue: a livello di Main dichiariamo :

```
static ManualResetEvent evtSyncronized;
static int RunningThreads;
```

A livello di classe con le Routine di elaborazione aggiungiamo poi la seguente variabile:

```
public int SyncID;
```

Infine avremo:

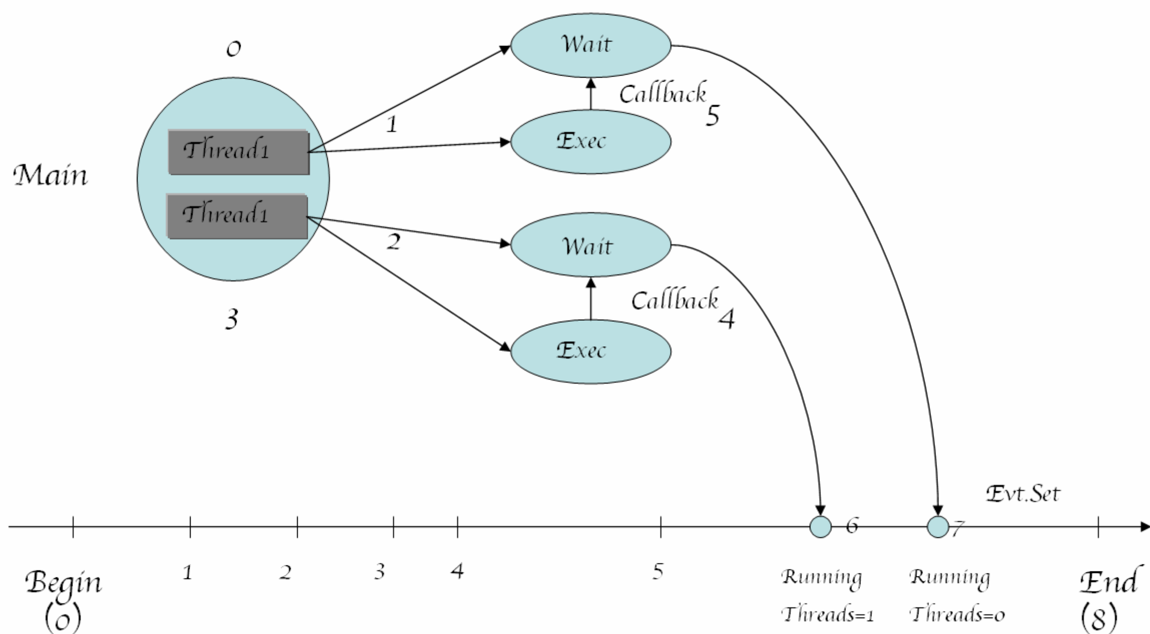
```
static void InvokeSyncRoutine(Object stateInfo)
{
        SyncRoutine objSR = (SyncRoutine)stateInfo;
        objSR .Exec();
        evtSyncronized.WaitOne();
}

public void CallbackResult(int ID, string RetMsgSyncro)
{
        RunningThreads--;
        StoreMessages(ID, RetMsgSyncro);
        if (RunningThreads==0)
        {
            UpdateFormControlsCrossThread();
            evtSyncronized.Set();
        }
}
```

Spiegazioni:

Il SyncID è semplicemente l'ID del singolo aggiornamento, ovvero quello che permette di targare l'update sulla tabella n-esima e quindi distinguerlo dalle altre. In pratica questo parametro viene passato alla SyncRoutine e questa semplicemente lo ripasserà indietro. Ad elaborazione completata il main (con la routine StoreMessages()) metterà il singolo messaggio nell'ordine giusto.

In questo schema invece di aspettare la fine di tutti i thread del pool con una istruzione tipo WaitHandle.WaitAll inserita a livello di Main come prevederebbero gli esempi nella documentazione Microsoft si fa qualcosa di diverso. Ovvero: si utilizza una variabile RunningThreads che inizialmente contiene il numero di thread da aprire (ovvero chiamate alle routines di update) : man mano che i Thread terminano questa variabile viene decrementata. Quando arriva a zero viene alzato un evento manuale evtSyncronized (vedi istruzione Set) che provoca la chiusura del Main Thread. Una spiegazione di quanto succede può essere data col seguente diagramma:



IL motivo di questa scelta è stato essenzialmente questo errore trovato nel caso di utilizzo di WaitAll :

"WaitAll for multiple handles on a STA thread is not supported."

Alcuni pareri illuminanti provenienti da forum vari quali i seguenti :

http://forums.msdn.microsoft.com/de-DE/netfxbcl/thread/eefc0160-987c-47e1-a41b-bbae96726ec4/

> Anyway, You can't use WaitHandle.WaitAll() on an STA thread as WaitAll is a blocking call , this is done to prevent the message pump to run. This is not allowed in Win32 and as such neither in .NET. Use one of the other Synchronization primitives like WaitOne or Thread.Join which do a limiited amount of pumping.

http://blogs.msdn.com/johnlee/archive/2007/07/10/waithandle-waitall-for-multiple-handles-on-a-sta-thread-is-not-supported.aspx

> While developing WinForm application, if you need to spawn multiple threads to work on different tasks and you need to wait all threads to complete ... then you might think to use the following approach:
>
> >>> Create an array of AutoResetEvent; use ThreadPool.QueueUserWorkerItem to spawn each task in different thread; then call WaitHandle.WaitAll(AutoResetEvent[]) to wait all threads to complete the task
>
> WinForm app is attributed as [STAThread] and you will get "WaitAll For Multiple Handles on a STA Thread Is Not Supported" error ... what's the solution? Here is what I used for this scenario:
>
> >>> Create a ManualResetEvent; maintain a TaskCounter; use ThreadPool.QueueUserWorkerItem to spawn each task in different thread; Call ManualResetEvent.WaitOne(); Decrement the TaskCounter after each task is completed and Call ManualResetEvent.Set() when the TaskCounter == 0
>
> Special notes: In your DoTask(object stateInfo), you need to handle exception properly and make sure the TaskCounter is subtracted should there is an exception

Mi hanno convinto a cambiare strada.

Un'ultima spiegazione la merita la UpdateFormControlsCrossThread(). Questa routine semplicemente pone in un controllo della form il risultato dell'elaborazione. Se però si prova a risolvere il tutto con qualcosa tipo :

textbox1.text = "Elaborazione Ok";

dalla callback o da una routine da questa chiamata si ottiene un nuovo infausto errore legato all'utilizzo dei Thread. Per motivi di protezione un thread non può modificare oggetti di un altro. Per questo si è costretti a ricorrere ancora ai delegati. Aggiungeremo in pratica alla classe della form main il seguente codice:

```
delegate void UpdateFormDlg();

private BackgroundWorker UpdateFormBckWrk = new BackgroundWorker();

private void UpdateFormControlsCrossThread()
{
    string msg = "";
    try
    {
       UpdateFormDlg d = new UpdateFormDlg(UpdateForm);
             this.Invoke(d);
    }
    catch (Exception ex)
    {
        msg = ex.Message;
    }
}

private void UpdateFormBckWrk_RunWorkerCompleted(
    object sender,
    RunWorkerCompletedEventArgs e)
{
    UpdateForm();
}

public void UpdateForm()
{
    textbox1.Text="Elaborazione ok";
}
```

Per i dettagli cercare su Google qualcosa tipo "aggiornamento form thread" e/o dare un'occhiata ai seguenti links:

http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=1442884&SiteID=1
http://www.coders4fun.com/it/2007/06/19/modificare-form-thread/

Conclusioni

La differenza fra approccio single thread e multi thread si mostra intanto per la mancanza dello sgradevole effetto di congelamento della form.

Il vantaggio principale è però in termini di migliore utilizzazione delle risorse hardware della macchina, come si capisce dai seguenti grafici:

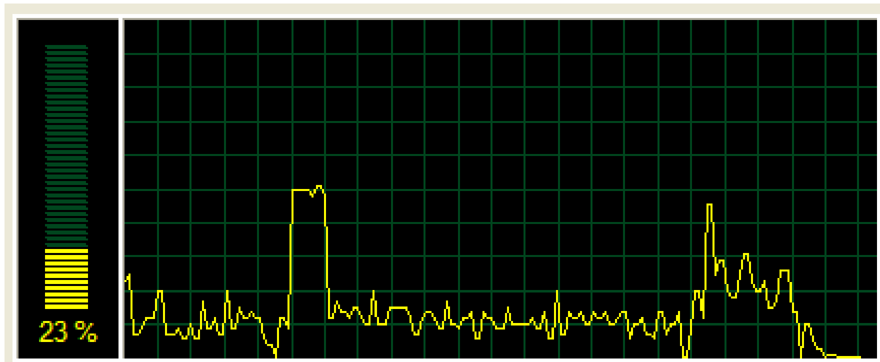


Figura 1 – Utilizzo CPU versione "single thread"

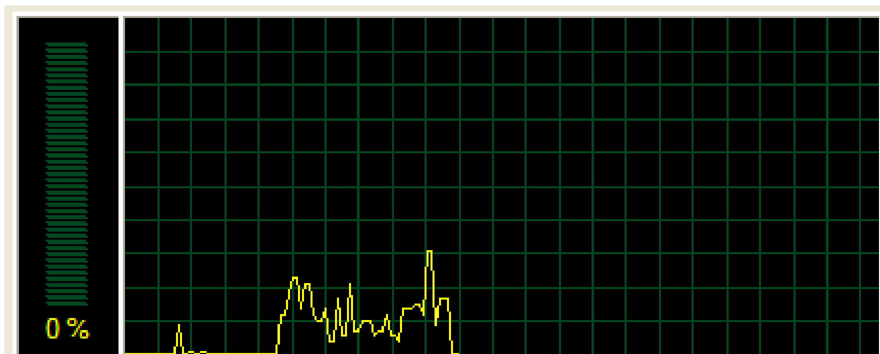


Figura 2 – Utilizzo CPU versione "multithread"

La prima differenza che salta all'occhio è quella della durata del processo. Fra parentesi anche la percentuale di impiego della CPU migliora, cosa che un po' mi lascia dubbioso.

Probabilmente perché in questa applicazione la risorsa critica è l'hard disk e nel primo caso il calcolatore deve prelevare, mettere in coda e elaborare mentre nel secondo i dati arrivano e vengono subito elaborati.

La maggiore efficienza si paga in una maggiore complessità di programmazione, con pesante utilizzo di classi aggiuntive e delegati.