

La Classe Monitor

Ver 0.0

C.Scozzafava

25/07/2009

www.southerlies.eu

southerlies.eu@gmail.com

La classe Monitor è un oggetto di sincronizzazione almeno in apparenza simile ad una Sezione Critica (Critical Section) ma le cui funzionalità ulteriori offrono degli interessanti spunti per comporre degli oggetti di basilare importanza in numerosissime applicazioni informatiche: le macchine a stati finiti (aka FSM – Finite States Machine).

La documentazione ufficiale pubblicata da Microsoft (Msdn: <http://msdn.microsoft.com/en-us/library/system.threading.monitor.aspx>) non sembra dare il dovuto risalto nè tantomeno riuscire ad essere esauriente su quali siano le possibilità di utilizzo della classe Monitor e quale la logica d'uso delle sue funzionalità ed in particolare dei metodi *Wait*, *Pulse* e *PulseAll*.

In questa Nota proverò a descrivere un uso corretto della classe Monitor così come implementata nel namespace System.Threading (.Net Framework da 1.1 a 4.0, includendo le Parallel Extensions); a conclusione verranno presentati degli esempi di possibili usi di un Monitor per la realizzazione di una macchina a stati finiti utilizzabile in un contesto di esecuzione multi-thread.

Monitor come Critical Section

Le funzionalità meglio note (e meglio documentate) della classe Monitor sono legate ai metodi *Enter*, *TryEnter* ed *Exit*.

Vediamo i prototipi di queste funzionalità seguiti da una breve descrizione a parole

```
public static void Enter ( object obj );  
public static bool TryEnter ( object obj );  
public static void Exit ( object obj );
```

| Funzionalità | Descrizione |
|-----------------|--|
| Enter | Acquisisce la proprietà dell'oggetto (parametro) <i>obj</i> . Garantisce che nessun altro thread possa entrare in possesso dello stesso <i>obj</i> quando quest'ultimo sia già posseduto. |
| TryEnter | Se un thread esegue il metodo Monitor.Enter su un <i>obj</i> già posseduto allora resterà bloccato finchè il possessore non esegua una Exit; Se un thread esegue il metodo Monitor.TryEnter su un <i>obj</i> già posseduto non sarà bloccato ma il valore restituito (booleano) sarà false . |
| Exit | Libera un <i>obj</i> precedentemente acquisito. Si noti che il thread che esegue Monitor.Exit deve essere lo stesso che ne possiede <i>l'obj</i> , ovvero che ha eseguito la Enter/TryEnter con successo. |

Limitandosi a usare solo questi tre metodi la classe Monitor ci mette a disposizione una classica

Critical Section.

Esempio:

```
class Program
{
    static object _sync = new object();

    static void mainThr_b()
    {
        Monitor.Enter(_sync);
        Console.WriteLine("Thr_b: Start work");
        Thread.Sleep(50);
        Console.WriteLine("Thr_b: End work\n");
        Monitor.Exit(_sync);
    }

    static void mainThr_a()
    {
        Monitor.Enter(_sync);
        Console.WriteLine("Thr_a: Start work");
        Thread.Sleep(50);
        Console.WriteLine("Thr_a: End work\n");
        Monitor.Exit(_sync);
    }

    static void Main(string[] args)
    {
        Thread thr_a;
        Thread thr_b;

        thr_a = new Thread(mainThr_a);
        thr_b = new Thread(mainThr_b);
        thr_a.Start();
        thr_b.Start();

        thr_a.Join();
        thr_b.Join();

        Console.ReadKey();
    }
}
```

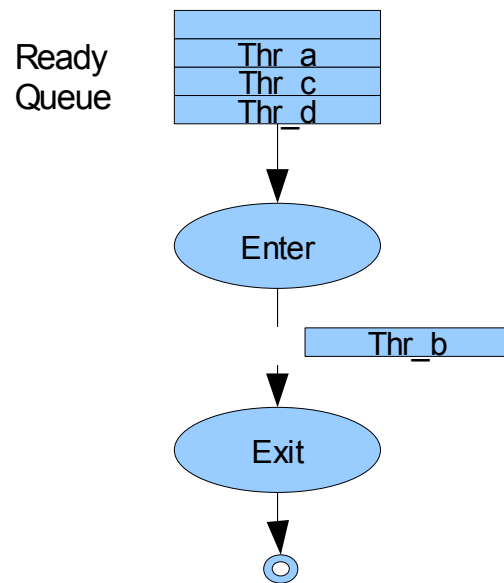
Si osservi che non è possibile determinare a priori quale dei due thread acquisirà l'oggetto `_sync`.

Monitor sotto la lente di ingrandimento

Prima di descrivere le restanti funzionalità del Monitor è opportuno provare a descrivere come tale oggetto tratti tutti quei thread che, arrivati ad eseguire una chiamata di `Enter`, vengono bloccati in attesa che l'oggetto conteso (*obj*) si liberi.

In effetti la classe Monitor mantiene delle strutture dati in modo che tutti quei thread che vengono bloccati sono, mano a mano, inseriti in una coda di tipo FIFO (Ready Queue).

Ciò significa che quando il possessore dell'oggetto *obj* passerà il turno (Exit) il thread che si trova sulla coda Ready in prima posizione potrà a sua volta diventare il nuovo possessore di *obj*.



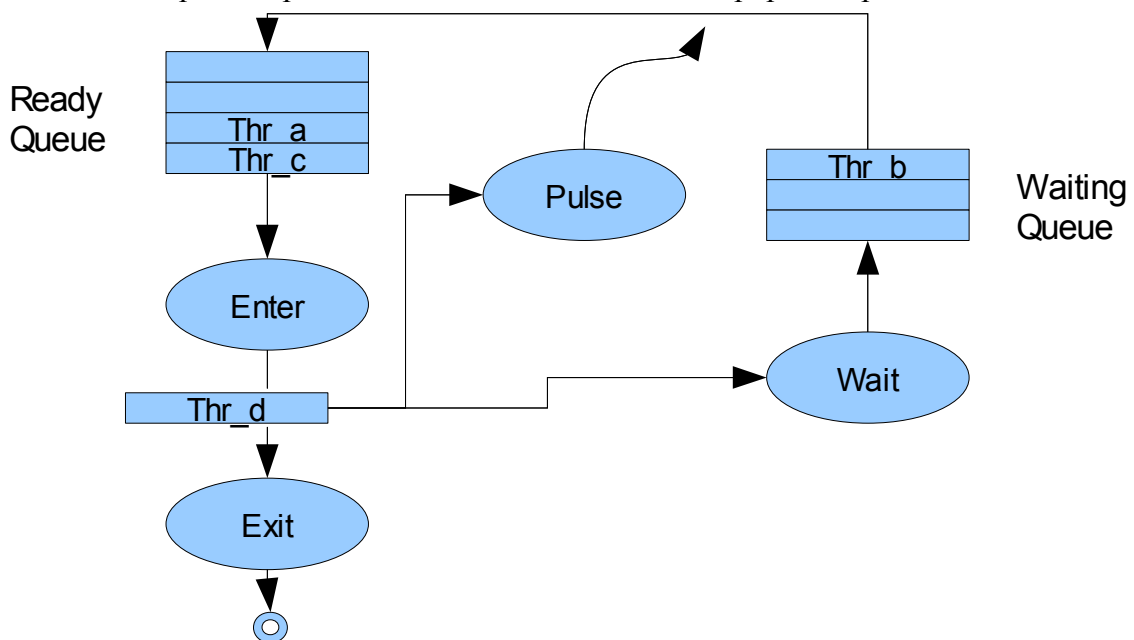
In figura è mostrato un thread Thr_b che è attualmente il possessore dell'oggetto *obj*. Altri thread, che hanno richiesto la acquisizione di *obj* sono stati messi in attesa nella coda Ready. Quando Thr_b deciderà di eseguire la Exit il primo della coda sarà libero di acquisire *obj*.

Tuttavia la Ready non è l'unica coda posseduta dal Monitor. Esso ha anche una seconda coda di thread detta Waiting.

Nella coda Waiting si trovano tutti quei thread che **dopo** aver acquisito *obj* hanno eseguito una chiamata a Wait; ed essi sono anche in attesa cioè sospesi o bloccati.

Inoltre tutti i thread che sono in Waiting potranno essere spostati in Ready solo quando un altro thread esegua una Pulse/PulseAll (si noti che un thread inserito in Waiting non può eseguire una Pulse perchè è sospeso).

Quindi uno schema più completo delle code e dei thread che le popolano può essere



Ecco che a questo punto dovrebbe iniziare ad essere più chiaro il significato ed il senso delle funzionalità di Monitor di cui non abbiamo ancora parlato: *Wait*, *Pulse* e *PulseAll*.

Monitor come oggetto di sincronizzazione 'attiva'

Vediamo i prototipi delle ulteriori funzionalità di Monitor ed una breve descrizione a parole

```
public static void Wait ( object obj );  
  
public static bool Pulse ( object obj );  
  
public static void PulseAll ( object obj );
```

| Funzionalità | Descrizione |
|-----------------|---|
| Pulse | Segnala che uno o più thread della Waiting Queue possono essere 'promossi' nella Ready Queue. |
| PulseAll | Pulse sposta il primo thread della Waiting Queue in Ready Queue, mentre PulseAll sposta tutti i thread. |
| Wait | Sospende il thread chiamante e lo inserisce in Waiting Queue. Il thread che chiama Wait deve essere in possesso dell'oggetto <i>obj</i> . (ovvero deve aver eseguito con successo una <i>Enter/TryEnter</i>). |

La situazione che risulta dopo l'illustrazione di queste nuove funzionalità è quindi tale da permetterci di considerare il Monitor come un elemento di coordinamento attivo invece che solo di mutua esclusione.

L'esempio successivo, benchè contenga un vincolo legato alla necessità che Thr_a parta prima di Thr_b, aiuta a comprendere come funzioni la coppia di funzionalità *Wait/Pulse*

Esempio:

```
class Program  
{  
    static object _sync = new object();  
    static bool _flag = true;  
  
    static void mainThr_b()  
    {  
        Monitor.Enter(_sync);  
  
        if(!_flag)  
        {  
            Console.WriteLine("Thr_b: goes in wait");  
            Monitor.Wait(_sync);  
            Console.WriteLine("Thr_b: exited the wait");  
        }  
    }  
}
```

```

Console.Write("Thr_b: Start work ");
Thread.Sleep(50);
Console.Write("Thr_b: End work\n");

Monitor.Pulse(_sync);
Monitor.Exit(_sync);
}

static void mainThr_a()
{
    Monitor.Enter(_sync);

    if(_flag)
    {
        Console.WriteLine("Thr_a: goes in wait");
        Monitor.Wait(_sync);
        Console.WriteLine("Thr_a: exited the wait");
    }

    Console.Write("Thr_a: Start work ");
    Thread.Sleep(50);
    Console.Write("Thr_a: End work\n");

    Monitor.Pulse(_sync);
    Monitor.Exit(_sync);
}

public static void Main(string[] args)
{
    Thread thr_a = new Thread(mainThr_a);
    Thread thr_b = new Thread(mainThr_b);

    //First start a, then b
    thr_a.Start();
    thr_b.Start();

    thr_a.Join();
    thr_b.Join();

    Console.ReadKey();
}
}

```

Il cui output è

```

Thr_a: goes in wait

Thr_b: Start work Thr_b: End work

Thr_a: exited the wait

Thr_a: Start work Thr_a: End work

```

Cioè il thread Thr_a, che viene avviato prima del Thr_b, entra nella sezione critica e verifica lo stato

di un flag; in base a questo valore decide o meno se sospendersi e spostarsi in Waiting Queue.

Il Thr_b è in attesa, sospeso in Ready Queue, di entrare nella sezione critica; ciò avviene non appena Thr_a si mette in wait.

Thr_b valuta il flag di stato ed in base al suo valore continua ad eseguire; subito prima di uscire dalla sezione critica provvede a segnalare (Pulse) alla Waiting Queue di spostare Thr_a in Ready Queue.

Thr_a è ora di nuovo attivo e la sua esecuzione riprende **a valle** della *Wait* con cui si era sospeso.

Quindi Thr_a quando riparte **non rivaluta lo stato** del flag! Per ottenere questo comportamento avremmo dovuto scrivere un `while` e non un `if` (**pattern while**).

```
static void mainThr_a()
{
    Monitor.Enter(_sync);

    while( _flag)
    {
        Console.WriteLine("Thr_a: goes in wait");
        Monitor.Wait(_sync);
        Console.WriteLine("Thr_a: exited the wait");
    }

    Console.Write("Thr_a: Start work ");
    Thread.Sleep(50);
    Console.Write("Thr_a: End work\n");

    Monitor.Pulse(_sync);
    Monitor.Exit(_sync);
}
```

Esempio: (pattern while)

```
static void mainThr_b()
{
    int _exec=0;
    while((_exec++)<10)
    {
        Monitor.Enter(_sync);

        while(!_flag)
            Monitor.Wait(_sync);

        Thread.Sleep(50);
        Console.Write("Thr_b: Start and End work\n");

        _flag = !_flag;
        Monitor.Pulse(_sync);
        Monitor.Exit(_sync);
    }
}
```

```

static void mainThr_a()
{
    int _exec=0;
    while((_exec++)<10)
    {
        Monitor.Enter(_sync);

        while(_flag)
            Monitor.Wait(_sync);

        Thread.Sleep(50);
        Console.Write("Thr_a: Start and End work\n");

        _flag = !_flag;
        Monitor.Pulse(_sync);
        Monitor.Exit(_sync);
    }
}

```

Il cui output è

```

Thr_b: Start and End work
Thr_a: Start and End work
Thr_b: Start and End work
Thr_a: Start and End work
Thr_b: Start and End work
Thr_a: Start and End work
...

```

Con il pattern while per la valutazione dello stato del flag scompare anche il vincolo sull'ordine di partenza dei thread (il che rende *corretta* la programmazione).

Nota:

L'uso delle chiamate *Enter/Exit* è anche codificato in un costrutto più elegante e robusto: il [lock](#).

I seguenti blocchi di codice sono esattamente identici

```

Monitor.Enter(_sync);
... //...
Monitor.Exit(_sync);

```

```

lock(_sync);
{
... //...
}

```

Usando un tool come Reflector è possibile verificare che dopo il codice IL generato dal costrutto lock viene trasformato in un *Enter/Exit*. L'utilità del lock è che introduce in maniera automatica l'uscita dalla sezione critica a fronte di eccezioni generate durante l'esecuzione. Si deve tener presente, infatti, che un'interruzione non gestita che avvenga all'interno del blocco *Enter/Exit* fa in

modo che l'oggetto `_sync` non venga rilasciato: ciò provoca il blocco di tutti gli altri thread che erano in attesa di acquisirlo.

Monitor, stato e pattern while

L'uso del pattern while consente di introdurre il concetto di stato in una classe che protegga le proprie strutture dati tramite dei Monitor. Una classe siffatta non è altro che una particolare FSM thread-safe.

Esempi: vedi codice allegato.

(to be continue...)