Thread: programmazione concorrente

Lo scopo di questa lezione è quello di fornire gli strumenti essenziali per potere implementare un'applicazione **multithread** in C#. I thread sono sotto-processi che possono essere eseguiti in parallelo o in serie. Essi collaborano per il raggiungimento di uno scopo comune, e, per questo motivo, possono anche condividere le stesse risorse (variabili, file, eccetera). La motivazione che risiede nella scrittura di un'applicazione multithread è legata alla possibilità di sfruttare tutti i **processori/core** ospitati all'interno delle nostre macchine (PC o smarthphone). Questo fa sì che le nostre applicazioni possano eseguire tanti compiti contemporaneamente, con un conseguente aumento della **velocità di calcolo**.
Ad esempio, per le applicazioni dotate di interfaccia grafica, si può pensare di eseguire porzioni di codice molto complesse *in background*, mentre i controlli dell'interfaccia rimangono attivi. Allo stesso modo, per un'applicazione server, utilizzare tanti thread consentirà di gestire separatamente le connessioni in ingresso. La documentazione ufficiale in materia di multithreading per C# è reperibile a **questo link**.

La classe Thread

Su C#, un thread è rappresentato dalla classe `Thread`, contenuta all'interno del namespace `System.Threading`. Per importarla, aggiungiamo in testa al codice la seguente clausola `using`:

```
using System.Threading;
```

Per inizializzare un thread, è sufficiente istanziare la classe `ThreadStart`, avente come parametro il nome del metodo che si intende eseguire come thread:

```
Thread t = new Thread(new ThreadStart(mythreadmethod));
```

La precedente riga istanzia un thread il cui nome è *t*, il cui codice da eseguire è quello contenuto all'interno di *myThreadMethod*.
Qualora *myThreadMethod* fosse parametrizzato, suggeriamo caldamente l'utilizzo della seguente tecnica per il passaggio dei parametri:

```
object myParams; //conterrà i parametri da passare in input al thread
... //codice per istanziare myParams
Thread t = new Thread(new ThreadStart(myThreadMethod(myParams)));
```

Di conseguenza, *myThreadMethod* dovrà necessariamente effettuare il cast dell'oggetto *myParams* per potere ricavare i parametri necessari all'esecuzione:

```
void myThreadMethod(object myparams)
{
... //codice per eseguire il casting di myparams and per leggerne il contenuto
}
```

I thread possono essere eseguiti sia in **background** che in **foreground**. L'unica differenza è che i thread in background vengono immediatamente terminati quando non vi sono più foreground thread in esecuzione. Per default, il processo padre (che è anche un thread) viene sempre eseguito in foreground.
I metodi principali della classe thread sono i seguenti:

- `Start()`: avvia l'esecuzione del thread. Dal momento in cui questo metodo viene invocato, il processo chiamante ed il thread chiamato eseguiranno le loro linee di codice indipendentemente l'uno dall'altro, e possibilmente in contemporanea;
- `Thread.Sleep(Int32 millis)`: sospende l'esecuzione del thread per il numero di millisecondi indicati;
- `Finalize()`: garantisce che tutte le risorse utilizzate dal thread verranno liberate dal garbage collector;
- `Abort()`: termina il thread, e genera un'eccezione `ThreadAbortException` che deve essere gestita dal processo padre;
- `Join()`: consente di bloccare l'esecuzione del thread chiamante finchè il thread per cui il metodo `Join` è stato invocato non viene terminato.

Le proprietà più salienti della classe Thread sono invece:

- `isAlive`: booleano che indica lo stato di esecuzione del thread;
- `isBackground`: indica se il thread viene eseguito in background;
- `ThreadState`: utile solamente in scenari di debugging, monitora lo stato di esecuzione del thread.

Sincronizzazione tra thread

L'esecuzione dei thread non avviene mai in un ordine prestabilito. Questo significa che più thread possono cercare di **accedere contemporaneamente alla stessa risorsa** e che non vi è mai garanzia che un thread termini prima di un altro. In genere si cerca di evitare al massimo la condivisione di risorse, ma alle volte non se ne può fare a meno. Qualora si vogliano eseguire diversi thread con una certa forma di organizzazione, proteggendo le risorse condivise, bisognerà ricorrere alla sincronizzazione.
Uno dei problemi principali che può insorgere quando si cerca di sincronizzare diversi thread è il **deadlock**. Esso consiste in una situazione in cui due o più processi attendono mutualmente che qualche altro thread faccia qualcosa; chiaramente, questo comporta una situazione di stallo. L'unica soluzione per prevenire situazioni di deadlock è la creazione di diagrammi di multithreading prima di iniziare la codifica.
C# mette a disposizione del programmatore diversi meccanismi per garantire la sincronizzazione dei thread. La parola chiave `lock()` viene utilizzata per indicare porzioni di codice che devono necessariamente essere eseguite senza interruzioni da altri thread. Il codice racchiuso al suo interno viene eseguito con esclusione reciproca tra i vari thread.
La sintassi è abbastanza semplice:

```
lock (object) {
... //codice protetto
}
```

Questa porzione di codice indica che tutti i thread che condividono un riferimento all'oggetto *object* devono fermarsi ad attendere l'esecuzione del codice del thread corrente.

Eventi di sincronizzazione

Talvolta risulta utile per un thread bloccare la sua esecuzione ed attendere che si verifichi un evento. A tale scopo, C# ha introdotto

i cosiddetti **eventi di sincronizzazione**. Un evento di sincronizzazione è un oggetto che può assumere due stati: **segnalato** e **non segnalato**. Le classi per gli eventi di sincronizzazione in C# sono `AutoResetEvent` e `ManualResetEvent`, con la differenza che `AutoResetEvent` commuta il suo stato da segnalato a non segnalato ogni volta che un thread si riattiva a seguito della verifica del suo stato segnalato.
Mentre `ManualResetEvent` deve essere manualmente riportato allo stato non segnalato, tramite chiamata del metodo `Reset()`. Per istanziare l'oggetto basterà dichiararlo come segue:

```
AutoResetEvent autoEvent = new AutoResetEvent(false);
```

I metodi salienti sono:

- `WaitOne()`: attende finchè l'evento di sincronizzazione non diventa segnalato. Quando l'evento di sincronizzazione viene segnalato, verrà immediatamente riportato alla condizione di non segnalato;
- `WaitAny(WaitHandle[])` e `WaitAll(WaitHandle[])`: consente di attendere almeno uno o tutti gli eventi di sincronizzazione contenuti nell'array `WaitHandle`;
- `Set()`: cambia lo stato da non segnalato a segnalato;
- `Reset()`: solo per i `ManualResetEvent`, cambia lo stato da segnalato a non segnalato.

Mutex

Uno dei metodi più classici per garantire l'accesso esclusivo ad una porzione di codice è l'utilizzo di una variabile di tipo **`Mutex`**. A differenza dall'utilizzo di `lock`, che limita il suo scope all'interno dell'applicazione, un oggetto di tipo `Mutex` può essere utilizzato per proteggere risorse a livello di sistema operativo. Dunque `Mutex` è un meccanismo più potente di `lock` e andrebbe utilizzato solo per la sincronizzazione tra thread appartenenti a diverse applicazioni.
Si dichiara come segue:

```
private Mutex mut = new Mutex();
```

Per verificarne lo stato si utilizza il metodo `WaitOne()` (come per gli eventi di sincronizzazione), ed il thread chiamante viene bloccato finchè la variabile non viene sbloccata, tramite chiamata del metodo `ReleaseMutex()`. Per evitare situazioni di deadlock, è possibile impostare un **timeout** dopo il quale il thread chiamante rinuncerà ad acquisire il controllo del `Mutex`:

```
if (mut.WaitOne(1000)) {
    //codice
}
```