Creare dei threads con Visual Basic

http://www.vbsimple.net/howto/ht_030.htm

Difficoltà: 4 / 5

Nello sviluppo di programmi di medie dimensioni si può rendere necessario lanciare una funzione che lavori *"per i fatti suoi"* e non blocchi l'esecuzione del codice fintanto che essa non è terminata.

Esistono due soluzioni per effettuare quest'operazione: la prima consiste nell'utilizzare un **Timer** insieme all'istruzione **DoEvents**. Il timer si dovrebbe occupare di lanciare la funzione ad intervalli regolari e l'istruzione DoEvents serve per permettere l'esecuzione di altri eventi mentre la funzione è in esecuzione.
Tuttavia questa soluzione, oltre ad essere molto scomoda, può generare svariati errori e risulta difficile da controllare.

L'altra soluzione consiste nel creare, come avviene in altri linguaggi, dei threads che si occupino di eseguire la funzione in maniera asincrona, non dipendente dal resto del programma. Sarà il sistema operativo che si occuperà di controllare il thread e il programmatore non si dovrà più occupare d'esso fino a quando non intende modificarlo, rallentarlo, velocizzarlo o terminarlo.
Si raccomanda di leggere la sezione dedicata alle Informazioni aggiuntive su Processi e Threads prima di approfondire la visione di questo codice.

Tuttavia, Visual Basic non permette nativamente l'utilizzo dei threads; sarà pertanto necessario utilizzare alcune funzioni API per creare e controllare il thread.
In questo esempio faremo utilizzo di un modulo di classe che permette l'istanza di nuovi threads e il pieno controllo d'essi mediante alcune proprietà e metodi della classe.

Pertanto, prima di vedere un esempio specifico, daremo uno sguardo alla classe ***clsThreads***.

```
1. Option Explicit
2.
3. Private Type udtThread
4.     Handle As Long
5.     Enabled As Boolean
6. End Type
7.
8. Private uThread As udtThread
9.
```

Alla riga 3 definiamo un nuovo tipo di dati definito dall'utente denominato **udtThread**. Al suo interno abbiamo due campi: *Handle* di tipo Long che conterrà l'handle del thread comandato dalla classe ed *Enabled* che indicherà se il thread è in esecuzione.

Alla riga 8 dichiariamo la variabile uThread di tipo udtThread. Tale variabile servirà per contenere i dati relativi al thread legato all'istanza.


```
10. Private Const CREATE_SUSPENDED As Long = &H4
11. Private Const THREAD_BASE_PRIORITY_IDLE As Long = -15
12. Private Const THREAD_BASE_PRIORITY_LOWRT As Long = 15
13. Private Const THREAD_BASE_PRIORITY_MAX As Long = 2
14. Private Const THREAD_BASE_PRIORITY_MIN As Long = -2
15. Private Const THREAD_PRIORITY_HIGHEST As Long = THREAD_BASE_PRIORITY_MAX
16. Private Const THREAD_PRIORITY_LOWEST As Long = THREAD_BASE_PRIORITY_MIN
17. Private Const THREAD_PRIORITY_ABOVE_NORMAL As Long = (THREAD_PRIORITY_HIGHEST - 1)
18. Private Const THREAD_PRIORITY_BELOW_NORMAL As Long = (THREAD_PRIORITY_LOWEST + 1)
19. Private Const THREAD_PRIORITY_IDLE As Long = THREAD_BASE_PRIORITY_IDLE
20. Private Const THREAD_PRIORITY_NORMAL As Long = 0
21. Private Const THREAD_PRIORITY_TIME_CRITICAL As Long = THREAD_BASE_PRIORITY_LOWRT
22.
```

La riga 10 definisce la costante CREATE_SUSPENDED che verrà utilizzata per creare un thread inizialmente fermo, non in esecuzione.
Le costanti definite alle righe 11-21 indicano le priorità assegnabili al thread in esecuzione.

```
23. Private Declare Function CreateThread Lib "kernel32" (ByVal lpThreadAttributes As
    Any, ByVal dwStackSize As Long, ByVal lpStartAddress As Long, lpParameter As Any,
    ByVal dwCreationFlags As Long, lpThreadID As Long) As Long
24. Private Declare Function SetThreadPriority Lib "kernel32" (ByVal hThread As Long,
    ByVal nPriority As Long) As Long
25. Private Declare Function GetThreadPriority Lib "kernel32" (ByVal hThread As Long)
    As Long
26. Private Declare Function SuspendThread Lib "kernel32" (ByVal hThread As Long) As
    Long
27. Private Declare Function ResumeThread Lib "kernel32" (ByVal hThread As Long) As
    Long
28. Private Declare Function TerminateThread Lib "kernel32" (ByVal hThread As Long,
    ByVal dwExitCode As Long) As Long
29.
```

Le funzioni API definite qui, servono per creare, cambiare la priorità, sospendere, ripristinare e terminare threads.

La prima funzione è la **CreateThread** e crea un nuovo thread che esegua il codice indicato dal puntatore alla funzione *lpStartAddress*. Le caratteristiche del thread saranno definite dal parametro *dwCreationFlags*. All'uscita la funzione riporta l'hande del thread creato.

La funzione **SetThreadPriority** permette di cambiare la priorità di un thread, mentre la **GetThreadPriority** viene utilizzata per ottenere la priorità del thread specificato.

La funzione **SuspendThread** serve per interrompere - ma non terminare - l'esecuzione di un thread. Analogamente, la **ResumeThread** ripristina la normale esecuzione del thread dopo la sua sospensione. In ultima analisi, la funzione **TerminateThread** termina l'esecuzione di un thread e ne distrugge l'handle.

```
30. Public Sub Initialize(ByVal lpfnBasFunc As Long)
31.     Dim lStackSize As Long
32.     Dim lCreationFlags As Long
33.     Dim lpThreadID As Long
34.     lStackSize = 0
35.     lCreationFlags = CREATE_SUSPENDED
36.     uThread.Handle = CreateThread(ByVal 0&, lStackSize, lpfnBasFunc, ByVal 0&,
    lCreationFlags, lpThreadID)
37.     If uThread.Handle = 0 Then MsgBox "Creazione del thread fallita!"
38. End Sub
39.
```

Il metodo principale di questa classe è **Initialize**. Esso richiede un puntatore a

funzione come parametro. Esso verrà utilizzato per creare il thread corrispondente.

Il cuore di questa funzione si trova alle righe 35 e 36. Inizialmente definiamo le modalità di creazione del thread. Nel nostro caso il thread verrà creato in maniera sospesa.

Alla riga 36 avviene la creazione del thread mediante chiamata alla funzione API *CreateThread*. Ad essa verranno passati vari parametri, molti dei quali impostati a 0. Il parametro principale è il puntatore alla funzione ricevuto dalla chiamata del metodo **Initialize**.

Se la creazione del thread è avvenuta in maniera corretta, il campo Handle della struttura uThread conterrà l'handle del thread creato. Se esso dovesse essere 0, il thread non sarà stato creato e sarà mostrato un messaggio di errore (riga 37).

Seguono un paio di proprietà che consentono di controllare l'esecuzione del thread creato dall'istanza della classe.

```
40. Public Property Get Enabled() As Boolean
41.     Enabled = uThread.Enabled
42. End Property
43.
44. Public Property Let Enabled(ByVal vNewValue As Boolean)
45.     If vNewValue = True Then
46.         ResumeThread uThread.Handle
47.     Else
48.         SuspendThread uThread.Handle
49.     End If
50.     uThread.Enabled = vNewValue
51. End Property
52.
```

La proprietà Enabled è utilizzabile in lettura ed in scrittura (Get e Let).
Essa determina se il thread è in esecuzione o meno.

La lettura della proprietà avviene semplicemente leggendo il valore dal campo *Enabled* della struttura **uThread**. La scrittura della proprietà comporta invece l'attivazione o la sospensione del thread. Dopo quest'operazione sarà necessario aggiornare il contenuto del campo Enabled di uThread.

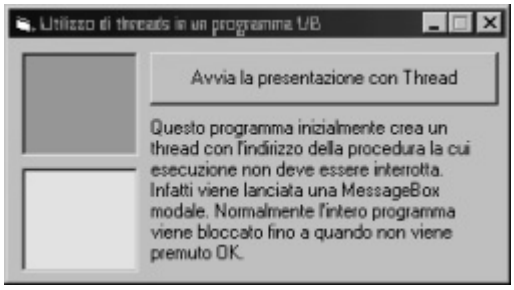
```
53. Public Property Get Priority() As Long
54.     Priority = GetThreadPriority(uThread.Handle)
55. End Property
56.
57. Public Property Let Priority(ByVal vNewValue As Long)
58.     Select Case vNewValue
59.         Case -2: Call SetThreadPriority(uThread.Handle, THREAD_PRIORITY_LOWEST)
60.         Case -1: Call SetThreadPriority(uThread.Handle,
    THREAD_PRIORITY_BELOW_NORMAL)
61.         Case 0: Call SetThreadPriority(uThread.Handle, THREAD_PRIORITY_NORMAL)
62.         Case 1: Call SetThreadPriority(uThread.Handle,
    THREAD_PRIORITY_ABOVE_NORMAL)
63.         Case 2: Call SetThreadPriority(uThread.Handle, THREAD_PRIORITY_HIGHEST)
64.     End Select
65. End Property
66.
```

Anche la proprietà Priority è in lettura e scrittura. La lettura utilizza la chiamata alla funzione *GetThreadPriority*; la scrittura invece utilizza la funzione *SetThreadPriority* per

cambiare la priorità del thread in esecuzione.

```
67. Private Sub Class_Terminate()
68.     Call TerminateThread(uThread.Handle, 0)
69. End Sub
```

La deallocazione dell'istanza comporta la sua distruzione e l'esecuzione dell'evento **Terminate**. Pertanto all'esecuzione di tale evento sarà necessario distruggere il thread creato e questo viene effettuato mediante la funzione *TerminateThread*.



Possiamo vedere l'implementazione della classe appena sviluppata. Inseriamo all'interno di un form due *PictureBox* di nome **Picture1** e **Picture2**, settando per esse due colori differenti nella proprietà *BackColor*.

Inseriamo anche un *CommandButton* di nome **AvviaThreads**. Abbiamo inserito anche una Label descrittiva ma essa non è effettivamente necessaria.

Il codice si compone esclusivamente di una routine: il click sul pulsante **AvviaThreads**.

```
 1. Option Explicit
 2.
 3. Private Sub AvviaThreads_Click()
 4.     Dim myThreadTop As New clsThreads
 5.     Dim myThreadBottom As New clsThreads
 6.
 7.     myThreadTop.Initialize AddressOf FlickerTop
 8.     myThreadTop.Enabled = True
 9.
10.     myThreadBottom.Initialize AddressOf FlickerBottom
11.     myThreadBottom.Enabled = True
12.
13.     MsgBox "I thread lanciati non sono bloccati..."
14.     Set myThreadTop = Nothing
15.     Set myThreadBottom = Nothing
16. End Sub
```

Alla righe 4 e 5 definiamo ed istanziamo due oggetti di classe clsThreads. Alla riga 7 creiamo il primo thread richiamando il metodo **Initialize**. Passeremo alla funzione il puntatore alla funzione ***FlickerTop*** (che per forza di cose deve risiedere all'interno di un modulo standard), che vedremo a breve. Creato il thread lo avviamo impostando la proprietà *Enabled* dell'istanza **myThreadTop**.

Operazione simile viene eseguita per la seconda istanza, **myThreadBottom**.
Inizializziamo il thread con il metodo **Initialize** e il puntatore alla funzione ***FlickerBottom*** e lo avviamo impostando la proprietà *Enabled*.

Per dimostrare l'effettiva attività dei threads richiamiamo una MessageBox (riga 13) che in situazioni normali blocca l'intera esecuzione del programma.
Ma, poiché abbiamo creato dei threads e lanciati, la visualizzazione della MessageBox blocca sì il programma, ma soltanto il codice che viene eseguito nel thread principale,

ovvero viene interrotta l'esecuzione della routine AvviaThreads_Click. Gli altri due thread saranno eseguiti regolarmente.

Alle righe 14 e 15 vengono deallocate e distrutte le due istanze. Così facendo saranno terminati i thread lanciati.

Prima di provare il programma dobbiamo definire le due funzioni da eseguire all'interno dei threads. Le due funzioni devono stare necessariamente all'interno di un modulo standard poiché l'operatore *AddressOf* che recupera l'indirizzo di una funzione richiede che la funzione risieda all'interno di un modulo standard.
Il codice è il seguente:

```
 1. Option Explicit
 2. Private Declare Function GetTickCount Lib "kernel32" () As Long
 3.
 4. Public Sub FlickerTop()
 5.     Static BgColor As Long
 6.     Dim lTick As Long
 7.     While 1 > 0
 8.         If BgColor <> &HFF& Then BgColor = &HFF& Else BgColor = &HFF00&
 9.         frmThread.Picture1.BackColor = BgColor
10.         frmThread.Picture1.Refresh
11.         lTick = GetTickCount
12.         While GetTickCount - lTick < 1250
13.         Wend
14.     Wend
15. End Sub
16.
```

Le due funzioni utilizzano la funzione API *GetTickCount*, vista peraltro in un altro HowTo, per effettuare un'attesa.

La prima funzione è la FlickerTop e provvede a cambiare il colore di sfondo di **Picture1** del form. Utilizzeremo una variabile statica, in maniera che all'uscita della funzione non venga azzerato il suo contenuto, di nome **BgColor**.

Alla riga 7 inizia un ciclo che non terminerà mai, ovvero sarà eseguito fintanto che 1 è maggiore di 0, ovvero a tempo indeterminato.

Alla riga 8 viene verificato il valore di BgColor: se esso è diverso da &HFF& (colore Rosso), sarà posto uguale a &HFF&, altrimenti sarà posto uguale a &HFF00& (verde). Adesso sarà possibile cambiare il colore della Picture1.
Alla riga 10 viene forzato l'aggiornamento grafico della Picture1; senza di esso il programma una volta compilato non mostrerebbe i colori in maniera corretta.

Alla riga 11 viene ottenuto il numero di millisecondi dall'avvio di Windows. Segue un ciclo che si ripete fintanto che non siano passati 1250 millisecondi dall'ultimo cambio di colore; rappresenta una maniera alternativa per effettuare un'attesa.

Il ciclo si ripete infinitamente e prima di cambiare il colore attende 1250 millisecondi. Ad ogni fase imposta il colore in Rosso o Verde.

```
17. Public Sub FlickerBottom()
```

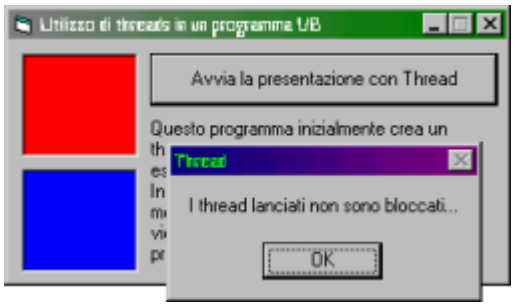


```
18.      Static BgColor As Long
19.      Dim lTick As Long
20.      On Error Resume Next
21.      While 1 > 0
22.          If BgColor <> &HFFFF& Then BgColor = &HFFFF& Else BgColor = &HFF0000&
23.          frmThread.Picture2.BackColor = BgColor
24.          frmThread.Picture2.Refresh
25.          lTick = GetTickCount
26.          While GetTickCount - lTick < 500
27.          Wend
28.      Wend
29.  End Sub
```

La funzione FlickerBottom esegue un'operazione del tutto identica alla precedente, eccetto per il fatto che i colori saranno &HFFFF& (Giallo) e &HFF0000& (Blu). Il colore sarà impostato nella **Picture2** ogni 500 millisecondi.

Sarà adesso possibile eseguire il programma, premere il pulsante e vedere i due threads lavorare, senza essere interrotti dalla MessageBox con il messaggio.

Ad intervalli regolari il colore delle due PictureBox sarà cambiato.



La soluzione proposta è molto geniale, ma non molto robusta. In altri linguaggi quali il Java l'operazione sarebbe notevolmente più semplice; purtroppo Visual Basic non contiene alcuna istruzione per permettere il richiamo o la gestione di threads multipli.

Attenzione!
È stato verificato che questo codice funziona correttamente con Visual Basic 5 ma per qualche ragione ancora sconosciuta, non funziona in progetti compilati con ***Visual Basic 6***. Se il progetto viene eseguito all'interno dell'IDE di VB6 funziona correttamente, ma una volta trasformato il progetto in file EXE autonomo, si generano errori di runtime non gestibili che forzano l'immediata terminazione del programma.

Codice originale di Peter Larsson
Modificato ed adattato da Fibia FBI
12 Marzo 2001
Rivisto e modificato il 7 Giugno 2001

Torna all'indice degli HowTo

