Informazioni aggiuntive su Processi e Threads

http://www.vbsimple.net/info/info_14.htm

Negli ambienti ***Win32*** (Windows 9x/NT/2000) è possibile eseguire più programmi comtemporaneamente. Il sistema dispone infatti di un sistema di *multitasking preemptive*, per il quale il tempo di esecuzione della CPU viene diviso in varie sezioni ognuna delle quali è detta **slice** (fetta); ogni sezione ha la durata approssimativamente di 20 millisecondi.

Il sistema, pertanto, assegna un certo numero di *slices* ai vari **task** (compiti) in base alla loro priorità rispetto agli altri compiti da eseguire. Questa separazione dei lavori *"illude"* l'utente che il sistema stia eseguendo più lavori contemporaneamente.

Ogni programma in esecuzione è detto **Processo** e dispone di un suo spazio di indirizzamento virtuale privato, che conterrà il codice del processo, i suoi dati ed altre risorse di sistema, quali i files, i tubi (pipe) e le periferiche di input e output.
Il processo crederà di avere a disposizione tutto il sistema, così come una grossa porzione di memoria (fino a 2 GB), persino maggiore della memoria fisica installata nel computer.
Un processo può anche creare ed allocare nuovi processi.

Ogni processo viene avviato con un singolo **Thread** in esecuzione, ma questo può creare nuovi threads e fargli svolgere diverse mansioni. Il sistema allocherà le varie slices di CPU ai singoli threads, in base alla loro priorità di esecuzione.

Alla scadenza del tempo di esecuzione assegnato al thread, il sistema sospende il thread, ne memorizza il contenuto nello spazio di indirizzamento del processo ed avvia l'esecuzione di un altro thread. Al termine dell'esecuzione di tutti i thread sospesi, viene riavviato il primo thread, dando così l'apparenza di sistema multitasking.
Il contenuto del singolo thread include il set di registri macchina, lo stack e la posizione in cui il thread è stato sospeso. Tutti i threads all'interno di un processo possono utilizzare tutti i dati globali all'interno dello spazio di indirizzamento virtuale e le risorse di sistema a disposizione del processo.

La priorità del thread è data dalla somma di tre valori:

1. Classe di priorità del processo:
   - *High* - Alta
     Per effettuare operazioni in cui la velocità è importante per assicurare un corretto funzionamento del programma, ad esempio l'invio di dati alla scheda video.
   - *Normal* - Normale
     La priorità di default, utilizzata dalla gran parte dei programmi.
   - *Idle* - Dormiente
     Classe di priorità molto bassa per eseguire operazioni nei tempi morti di elaborazione.
   - *Realtime* - Tempo reale

La priorità più alta possibile a disposizione del sistema. I programmi non dovrebbero utilizzare mai tale classe di priorità per non bloccare l'esecuzione di altri programmi. Viene utilizzata dai driver per le operazioni critiche e brevi quali il tempo di risposta del mouse o la scrittura dei dati di un programma di cache.

2. Livello di priorità all'interno della classe del processo:
   - *Lowest* - Il più basso
   - *Below normal* - Sotto il normale
   - *Normal* - Normale
   - *Above normale* - Sopra il normale
   - *Highest* - Il più alto

3. Acceleratore di priorità
   Assegnato dal sistema operativo in caso di richieste di input o output di un thread.

Ogni processo ed ogni thread viene identificato da un suo handle unico.
Un processo avvia inzialmente un thread e quest'ultimo si occuperà di avviare gli altri threads per svolgere il lavoro del processo.

È possibile verificare lo stato di esecuzione di un thread utilizzando la funzione API *GetExitCodeThread*. Fintanto che il thread è in esecuzione, la funzione restituisce il valore API ***STILL_ACTIVE*** (259). Il thread può essere sospeso o fatto ripartire attraverso un altro thread mediante l'utilizzo delle funzioni API *SuspendThread* e *ResumeThread*.

Un thread può essere terminato in quattro modalità:

- Chiamata della funzione API *ExitThread* da parte del thread in esecuzione.
- Ritorno della funzione del thread, che genera un *ExitThread* o *ExitProcess* implicito.
- Chiamata della funzione API TerminateThread da parte di un altro thread o di un altro processo.
- Chiamata alla funzione API *TerminateProcess* con l'handle del processo in cui il thread è eseguito, che effettua la chiusura del processo.

Dopo la chiusura del thread il richiamo della funzione *GetExitCodeThread* restituirà il codice di uscita della funzione eseguita dal thread.

Nella sezione HowTo è presente un esempio per creare e chiudere dei threads con Visual Basic.

Un processo, fintanto che è in esecuzione, restituisce alla chiamata *GetExitCodeProcess* il valore API **STILL_ACTIVE** (259). La chiusura del processo è invocata in quattro situazioni:

- Un thread qualunque del processo richiama la funzione API *ExitProcess*.
- Il thread primario del processo termina la sua esecuzione mediante ritorno. Questo

infatti genera una chiamata implicita alla funzione *ExitProcess*.
- Tutti i thread del processo terminano.
- Qualche thread ha richiamato la funzione *TerminateProcess* con l'handle del processo.

Alla chiusura del processo tutti i files lasciati aperti saranno chiusi e gli handle degli oggetti sono deallocati soltanto quando tutti i processi che li utilizzano chiudono tali handle.
Tuttavia, i processi richiamati dal processo appena chiuso non saranno automaticamente chiusi.

Nella sezione HowTo è presente un esempio per creare e chiudere un processo con Visual Basic.

Fibia FBI
30 Aprile 2001
