

Aprile 1999

**PROGRAMMAZIONE
IN AMBIENTE
MICROSOFT WINDOWS**

**Win32 con Visual C++
Aspetti avanzati**

Introduzione all'ambiente Windows
e strumenti di programmazione

*Ing. Stefano Riccio
stefaric@tiscalinet.it*

BIBLIOGRAFIA

 Per quanto riguarda la programmazione in 'C' utilizzando SDK:

- **Manuali Visual C++**
- **Manuali Microsoft SDK (Software Development Kit)**
- **Programmazione avanzata per windows - III edizione**
J. Rithcher - Mondadori informatica

 Altri testi utili:

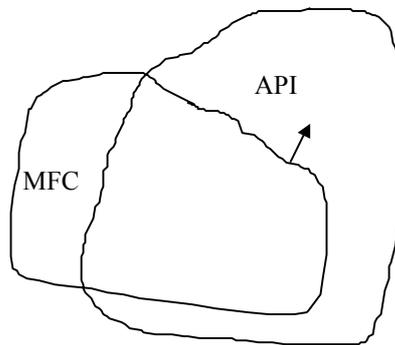
- **Programmare in windows. Gestione dell'input e uso delle risorse**
Petzold - Mondadori informatica
- **Programmare in windows. GDI, DDE e collegamenti**
Petzold - Mondadori informatica

Diversi livelli di utilizzo di Visual C++

- utilizzo programmazione ad oggetti usando MFC (documento-vista) e strumenti di ausilio al programmatore (Application Wizard, ClassWizard)
- utilizzo programmazione ad oggetti usando MFC con documento-vista
- utilizzo programmazione ad oggetti usando MFC senza documento-vista
- utilizzo programmazione ad oggetti (C++) senza usare Microsoft Foundation class (MFC)
- utilizzo SDK (programmazione in C) con API Win 32
- utilizzo SDK (programmazione in C) con API Win 16

Il corso sara' rivolto all'utilizzo delle API versione WIN32.

Relazione tra API e MFC



- Le API sono un insieme di funzioni raccolte in librerie a collegamento dinamico (DLL). Esse rappresentano l'interfaccia tra le applicazioni e il sistema operativo.
- MFC rappresenta una gerarchia di classi che incorpora parte delle API.

Windows non e' un sistema ad oggetti, quindi e' stata creata una libreria (API) che rappresentano le funzioni del sistema operativo. Se MFC riuscirà in futuro ad inglobare tutte le API e sara' fornita gratuitamente con Windows si potra' effettivamente parlare di Windows come Sistema operativo object oriented.

MFC purtroppo non copre tutte le api di windows, quindi il programmatore esperto spesso ha bisogno di funzioni non contemplate nelle MFC e deve utilizzare le API di windows. Ma e' pur vero che lavorare esclusivamente con le API al giorno d'oggi non ha piu' molto senso perche' la gerarchia di classi rende argomenti complessi molto semplici e veloci da implementare (grazie anche all'utilizzo di wizard). Quindi e' necessario conoscere le API e la gerarchia MFC e sapere scegliere quando utilizzare l'una o l'altra.

API WIN32

Esistono 3 API :

- Win 16 per i sistemi a 16 bit
- Win 32 per i sistemi a 32 bit
- in ambiente Windows 3.1 esistono le Win32s, le quali convertono i parametri di alcune funzioni da 32 bit a 16 bit ma sotto continuano ad usare Win 16. Win32s e' ormai superata.

La WIN32 completa e' contenuta in Windows NT. In Windows 95/98 e' contenuta una versione non completa di Win32 (ad esempio mancano alcune funzioni di I/O asincrono, alcune funzioni di sicurezza ecc..).

Non e' ancora chiara la versione di Win32 installata su Windows CE.

Il cuore delle API e' formato da 3 DLL:

- 1) Kernel : memory manager, scheduler, loader
- 2) User: sistema di windowing
- 3) GDI: graphics device interface

In seguito sono state create una infinita' di DLL, ognuna delle quali concerne uno specifico argomento. Continuamente escono nuove versioni e nuove DLL riguardanti innovazioni tecnologiche.

Facciamo alcuni esempi di ultime estensioni:

- WinSocket per la comunicazione TCP-IP (o networking)
- MAPI per la posta elettronica
- RAS accesso remoto
- ODBC accesso ai database via SQL
- backup su nastro
- DirectX per la grafica 3D
- ecc...

Aprile 1999

IL MULTITASKING

*Ing. Stefano Riccio
stefaric@tiscalinet.it*

PROCESSI - THREAD

Si definisce PROCESSO l'istanza di un programma in esecuzione. Un processo possiede 4 GB di spazio di indirizzamento.

Anche le DLL caricate dal processo condividono lo stesso spazio di indirizzamento.

Affinche' un processo realizzi qualche operazione e' necessario che possieda un THREAD. Infatti ogni processo ha almeno un thread (detto thread primario) creato automaticamente.

Win32 supporta due tipi di processi: GUI (graphical user interface) e CUI (console user interface).

I processi GUI devono iniziare con la funzione WinMain.

I processi possono creare altri processi attraverso la funzione CreateProcess. Vecchie funzioni di Win16 come WinExec e LoadModule richiamano in realta' CreateProcess.

In realta' la possibilita di creare processi permette di eseguire programmi EXE richiamandoli direttamente dal codice di un programma (Operazione gia' molto semplice in passato).

Estremamente piu' interessante e' la possibilita' di creare processi contenenti piu' thread (vedremo piu' avanti).

Un thread descrive un percorso di esecuzione all'interno di un processo. Questo aspetto e' nuovo nel mondo dei personal computer con sistemi operativi Windows che derivano dal mondo dos.

I Thread e quando farne uso

Moltissimi sono i casi in cui e' possibile usare i thread. Ad esempio ricalcolare un foglio elettronico; stampare un documento in background.

Minori sono i casi in cui e' consigliato usare i thread, i quali apportano una notevole complessita' al software e una forte difficolta' nella loro gestione. Ad esempio il caso classico della stampa in background non e' esente da problemi.

Ogni Thread possiede un proprio stack nel quale memorizzare le variabili automatiche (ricavato dallo spazio di indirizzamento del processo che lo ha generato). Quando si usano variabili statiche e globali, piu' thread possono accedervi contemporaneamente.

Ogni Thread possiede una propria area di memoria utile a salvare il contesto rappresentato dai registri della CPU. Questo al fine di realizzare un multitasking a partizione di tempo. Infatti la CPU assegna a turno ad ogni task un quantum di tempo di utilizzo della CPU.

I thread vengono creati attraverso la funzione CreateThread.

Il Multitasking e le code dei messaggi

Ciascun processo ha almeno un thread.

La CPU distribuisce il tempo tra i thread e non tra i processi.

Il numero di thread che possono essere creati non ha limite se non quello delle risorse del computer.

In Win32 quando viene creato un thread, esso e' considerato come un thread "operaio", in altre parole il sistema tenta di mantenere il thread a basso livello, in modo da occupare poche risorse. Ma appena il thread richiama una funzione User o GDI (ad esempio CreateWindow), esso diventa un thread di interfaccia utente.

Ai thread di interfaccia utente viene associata una struttura (THREADINFO). Tale struttura non e' documentata dalla Microsoft e serve per gestire delle code di messaggi per il thread.

La struttura THREADINFO e' molto complessa, occupa molte risorse, ma permette al thread di spedire e ricevere messaggi.

THREADINFO contiene:

- 1- un puntatore ad una coda dinamica di messaggi esposti (ossia i messaggi che il thread spedisce all'esterno).
- 2- un puntatore ad una coda dinamica di messaggi di input (ossia i messaggi che arrivano al thread dalla tastiera, mouse ecc..).
- 3- un puntatore ad una coda dinamica di messaggi in arrivo da altri thread.

Codice per utilizzare i Thread

- Per creare un thread si usa la funzione

```
HANDLE CreateThread( LPSECURITY_ATTRIBUTES Ipsa,  
                    DWORD cbStack, //grandezza stack  
                    LPTHREAD_START_ROUTINE lpStartAddr, //indirizzo della  
                                                            funzione  
                    LPVOID lpvThreadParm,  
                    DWORD fdwCreate,  
                    LPWORD lpIDThread)
```

- Ogni thread inizia a girare con una funzione che il programmatore deve specificare, la funzione ha la seguente forma

```
DWORD WINAPI <Nome> (LPVOID lpvThreadParm)
```

lpvThreadParm viene passato dalla CreateThread

- Un thread puo' terminare in 4 modi:

- 1) Terminando la funzione thread associata, in questo caso il thread ritorna cio' che viene restituito dalla funzione thread stessa

- 2) chiamando *ExitThread(UINT fuExitCode)* dall'interno della funzione associata al thread. In questo caso la terminazione viene segnalata agli altri, i quali vengono a conoscenza del fatto che lo stack viene deallocato
- 3) Un altro thread chiama *TerminateThread(HANDLE hThread, DWORD dwExitCode)*, in questo caso non viene liberato lo stack del thread che termina, poiché altri thread potrebbero referenziare ancora tale memoria
- 4) Il processo contenente il Thread termina, il che implica la terminazione di tutti i thread del processo

Quando un thread termina vengono deallocate tutte le risorse di proprietà del thread, ma non quelle in comune con il processo.

Un altro thread può controllare se il thread è terminato mediante

```
BOOL GetExitCodeThread(HANDLE hThread, LPWORD lpdwExitCode)
```

Il thread può conoscere a quale processo appartiene con

```
HANDLE GetCurrentProcess(VOID)
```

All'interno della funzione thread si può venire a conoscenza dell'handle del thread mediante:

```
HANDLE GetCurrentThread(VOID)
```

La sincronizzazione dei Thread

In un ambiente in cui girano in concomitanza più thread, diventa importante essere capaci di sincronizzare le loro attività.

Esistono 5 principali oggetti per la sincronizzazione:

- 1) le sezioni critiche
- 2) i mutex
- 3) i semafori
- 4) gli eventi
- 5) i temporizzatori di attesa (solo per Windows NT)

In generale un thread si sincronizza con un altro thread mettendosi a dormire. Mentre un thread dorme la CPU non gli assegna tempo. Tuttavia prima di mettersi a dormire il thread comunica al sistema operativo quale "evento speciale" deve verificarsi per riattivare l'esecuzione.

1) LE SEZIONI CRITICHE

La sezione critica è una piccola sezione di codice che richiede l'accesso esclusivo ad alcuni dati condivisi prima che il codice possa essere eseguito. Le sezioni critiche consentono a un solo thread alla volta di ottenere l'accesso a una zona di dati.

Per creare una sezione critica, bisogna innanzitutto assegnare nel proprio processo una struttura `CRITICAL_SECTION`. L'allocazione di questa struttura deve essere globale in modo che vi possano accedere più thread.

La sezione critica va inizializzata con

```
VOID InitializeCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

Si puo' accedere alla sezione critica richiedendone l'accesso

```
VOID EnterCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

e rilasciandone l'accesso esclusivo il prima possibile

```
VOID LeaveCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

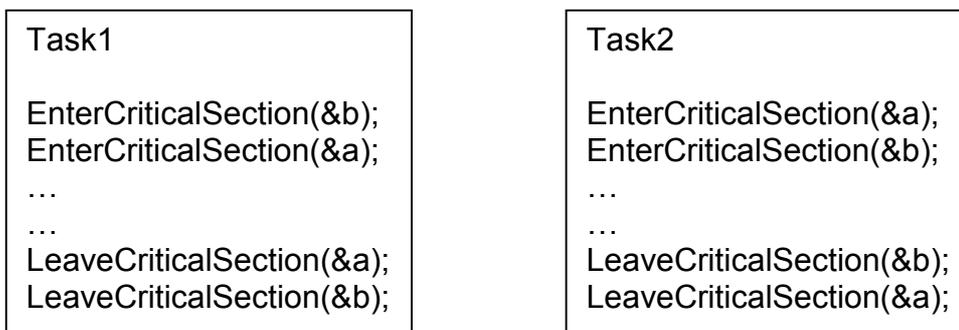
Quando un thread tenta di accedere, se la sezione critica e' gia' occupata viene messo in sonno in attesa che si liberi la sezione critica stessa.

Alla fine tutte le sezioni critiche devono essere liberate con

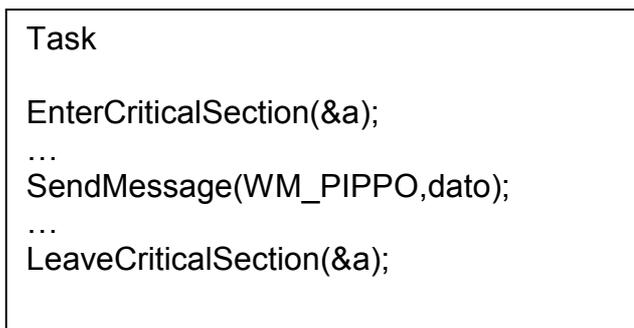
```
VOID DeleteCriticalSection( LPCRITICAL_SECTION lpCriticalSection);
```

Le sezioni critiche hanno dei problemi:

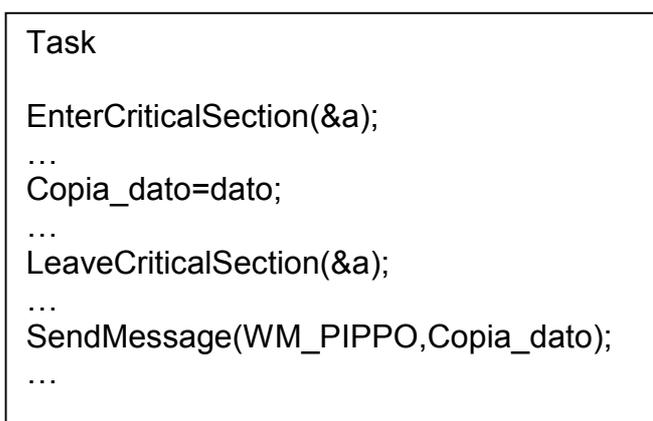
e' possibile che il programmatore crei una situazione di Deadlock, ad esempio



Altro problema e' l'invio di messaggi durante l'occupazione di una sezione critica



In questo caso non e' possibile prevedere il tempo di esecuzione di SendMessage, quindi si tengono occupati i dati per un tempo indefinito, e' meglio fare nel seguente modo:



In questo caso occupo la sezione critica solo per fare una copia dei dati.

N.B : Solo per Windows NT 4 e' disponibile la funzione TryEntryCriticalSection che permette di sapere se una sezione critica e' occupata oppure no (in pratica e' simile a EntryCriticalSection, solamente non attende la liberazione della sezione critica).

GLI OGGETTI KERNEL

E' possibile sincronizzare le applicazioni in funzioni di particolari eventi relativi alla macchina o ad altri processi.

Gli oggetti kernel sono:

- processi
- thread
- input di tastiera
- notifiche di variazioni dei file
- mutex
- semafori
- eventi (a ripristino automatico e manuale)
- temporizzatori di attesa (solo Windows NT)

Ciascun oggetto puo' essere in ogni momento in uno dei due seguenti stati:

-segnalato

-non segnalato

Per sincronizzare i thread, e' possibile metterli in sonno fino a quando uno degli oggetti suddetti diventa segnalato.

I thread si servono di due funzioni per mettersi in sonno mentre attendono che gli oggetti kernel diventino segnalati:

```
DWORD WaitForSingleObject( HANDLE hHandle, //attende un oggetto
                           DWORD dwMilliseconds )
```

```
DWORD WaitForMultipleObjects( DWORD nCount,
                              CONST HANDLE *lpHandles, //attende una lista di oggetti
                              BOOL bWaitAll,
                              DWORD dwMilliseconds );
```

N.B: Queste due funzioni hanno degli effetti sugli oggetti. Per gli oggetti processo e thread non vi e' alcun effetto, mentre per gli altri dopo essere diventati SEGNALATI vengono subito messi in stato di NON SEGNALATO.

2) I MUTEX

I mutex sono simili alle sezioni critiche, ma possono servire per sincronizzare l'accesso ai dati fra piu' processi (cosa non possibile con le sezioni critiche).

Si puo' creare un mutex con la funzione

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTES lpMutexAttributes,
                   BOOL bInitialOwner,
                   LPCTSTR lpName );
```

Al momento della creazione viene creato un oggetto kernel di Windows di tipo mutex denominato *lpName* (questo nome e' importante per condividere i mutex tra i processi). La funzione restituisce un handle univoco per il sistema operativo.

Ogni processo puo' richiamare `CreateMutex` con lo stesso nome per condividere lo stesso Mutex.

Con la funzione `GetLastError` e' possibile conoscere se un mutex esiste gia' oppure no. In alternativa e' possibile utilizzare

```
HANDLE OpenMutex( DWORD dwDesiredAccess,  
                 BOOL bInheritHandle,  
                 LPCTSTR lpName );
```

Viene restituito `NULL` se il mutex non esiste.

Per rilasciare il mutex

```
BOOL ReleaseMutex( HANDLE hMutex);
```

N.B: ricordarsi di creare i mutex prima dei thread per essere sicuri della loro esistenza prima di usarli.

3) I SEMAFORI

I semafori vengono utilizzati per il conteggio delle risorse. Essi forniscono al thread la capacita' di richiedere il numero di risorse disponibili.

Ad esempio se un computer ha tre porte seriali, in ogni momento solo tre thread possono accedere alla porta seriale. Si puo' creare un semaforo con il contatore impostato a 3. Il semaforo e' segnalato quando il conteggio e' maggiore di 0 e non segnalato quando il contatore e' 0.

Il semaforo si crea con

```
HANDLE CreateSemaphore( LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  
                      LONG lInitialCount,  
                      LONG lMaximumCount,  
                      LPCTSTR lpName );
```

Anche in questo caso si puo' condividere il semaforo, attraverso il nome *lpName*, tra processi. Per gli altri processi e' possibile richiamare `CreateSemaphore` oppure usare

```
HANDLE OpenSemaphore( DWORD dwDesiredAccess,  
                    BOOL bInheritHandle,  
                    LPCTSTR lpName );
```

Per rilasciare il semaforo

```
BOOL ReleaseSemaphore( HANDLE hSemaphore,  
                      LONG IReleaseCount,  
                      LPLONG lpPreviousCount );
```

4) GLI EVENTI

Esistono due tipi di eventi:

- eventi a ripristino manuale
serve per segnalare contemporaneamente a piu' thread che un'operazione e' stata completata
- eventi a ripristino automatico
serve per segnalare ad un unico thread che un'operazione e' stata completata

Un evento viene utilizzato da un thread ad esempio per segnalare la fine di un'operazione di inizializzazione.

Il thread di inizializzazione crea un evento e lo imposta sullo stato di NON segnalato, quindi comincia ad inizializzare i dati. Quando finisce imposta l'evento sullo stato di segnalato. Automaticamente un thread con il compito di eseguire un'operazione sui dati viene svegliato dalla segnalazione di un evento.

Per creare un evento:

```
HANDLE CreateEvent( LPSECURITY_ATTRIBUTES lpEventAttributes,  
                   BOOL bManualReset,  
                   BOOL bInitialState,  
                   LPCTSTR lpName );
```

Anche in questo caso un altro processo puo' usufruire del medesimo evento usando la stessa CreateEvent oppure:

```
HANDLE OpenEvent( DWORD dwDesiredAccess,  
                 BOOL bInheritHandle,  
                 LPCTSTR lpName );
```

Gli eventi a ripristino manuale non vengono automaticamente impostati sullo stato non segnalato dalle funzioni WaitForSingleObject e WaitForMultipleObjects.

Nel caso dei mutex, quando un thread chiama una di queste funzioni si mette a dormire fino a quando il mutex diventa segnalato e di conseguenza ripristinano automaticamente il mutex nello stato di non segnalato.

Cio' e' importante per garantire che un solo thread si risvegli in conseguenza della segnalazione del mutex.

Per gli eventi a ripristino manuale la cosa e' diversa: potrebbero esserci diversi thread in attesa, in questo caso tutti i thread si risvegliano per elaborare le informazioni.

Un thread puo' impostare l'evento allo stato di segnalato con

```
BOOL SetEvent( HANDLE hEvent);
```

Una volta che viene segnalato, questo rimane in tale stato fino a quando non viene ripristinato mediante:

```
BOOL ResetEvent( HANDLE hEvent);
```

4) I TEMPORIZZATORI DI ATTESA

Sono stati aggiunti con Windows NT 4. (NON FUNZIONANO IN WINDOWS 95)
Sono oggetti kernel che segnalano la loro presenza a una determinata ora e/o ad intervalli regolari.

Per creare un temporizzatore e' sufficiente chiamare

```
HANDLE CreateWaitableTimer( LPSECURITY_ATTRIBUTES lpTimerAttributes,  
                             BOOL bManualReset,  
                             LPCTSTR lpTimerName );
```

Anche in questo caso un altro processo puo' usare

```
HANDLE OpenWaitableTimer( DWORD dwDesiredAccess,  
                           BOOL bInheritHandle,  
                           LPCTSTR lpTimerName );
```

Una volta acquisito l'handle di un temporizzatore di attesa, si puo' impostare il temporizzatore chiamando:

```
BOOL SetWaitableTimer( HANDLE hTimer,  
                       const LARGE_INTEGER *pDueTime,  
                       LONG lPeriod,  
                       PTIMERAPCROUTINE pfnCompletionRoutine,  
                       LPVOID lpArgToCompletionRoutine,  
                       BOOL fResume );
```

Questa funzione e' piuttosto complessa e ricca di parametri.

I/O ASINCRONO

Per crear un canale di comunicazione con un dispositivo esterno si usa:

`HANDLE CreateFile`

Specificando il flag `FILE_FLAG_OVERLAPPED` e' possibile compiere una lettura ASINCRONA. Questo significa che dopo aver richiamato il codice `ReadFile` si continua l'esecuzione senza attendere la fine del caricamento dei dati.

Per leggere dati da un dispositivo esterno (quale una porta seriale) si deve usare:

`ReadFile`

Ci sono 4 tecniche per segnalare la fine del caricamento dei dati (vediamo solamente le prime due):

1) segnalazione di un oggetto kernel dispositivo

Si conserva l'handle restituito dalla `create file`. Dopo aver eseguito una `ReadFile` si mette in attesa un thread sulla segnalazione dell'handle del file.

`WaitForSingleObject(...,handleFile,...)`

Quando il sistema ha finito di caricare i dati rende l'handle del file SEGNALATO e il thread in attesa puo' elaborare i dati.

E' possibile controllare se l'operazione ha avuto buon fine con la funzione:

`GetOverlappedResult`

2) segnalazione di un oggetto evento

Questo serve quando voglio gestire piu' richieste di I/O contemporaneamente. In questo caso e' possibile associare al caricamento dei dati un oggetto kernel di tipo evento che permetta di risvegliare un thread di elaborazione

PRIORITA' IN WINDOWS NT

E' possibile modificare la priorita di un processo figlio con

`SetPriorityclass(handle del processo,...)`

Ogni thread viene creato con priorita' eguale a quella del processo padre.
Si puo' modificare la priorita di un thread con

`SetThreadPriority(handle del thread,...)`

Aprile 1999

LA GESTIONE DELLA MEMORIA IN WINDOWS 95 - NT

*Ing. Stefano Riccio
stefaric@tiscalinet.it*

L'ARCHITETTURA DELLA MEMORIA

In Win32, ogni spazio di indirizzamento virtuale di un processo e' di 4 GB. Ossia tutto lo spazio di indirizzamento offerto da un puntatore a 32 bit.

In MSDOS e in Windows 16 bit tutti i processi condividono un solo spazio di indirizzamento. Questo significa che ogni processo puo' scrivere e leggere la memoria degli altri, compreso il sistema operativo.

In Win32 invece ogni processo ha un proprio spazio di indirizzamento privato. In Windows NT anche la memoria del sistema operativo e' privata e protetta (cosa non vera in Windows 95).

E' chiaro che questa enorme quantita' di spazio indirizzabile non e' memoria fisica, ma virtuale.

Le zone

Quando viene creato un processo tutto lo spazio e' libero. Per usare porzioni dello spazio di indirizzamento e' necessario delle zone al suo interno, chiamando la funzione *VirtualAlloc*.

Ogni volta che si riserva una zona dello spazio di indirizzamento, il sistema garantisce che la zona stessa inizi a un limite esatto della granularita' di assegnazione. La granularita' dell'assegnazione puo' variare da una piattaforma ad un'altra'. In sistema X86 si tratta di 64KB. Inoltre le zone sono di dimensioni multiple esatte della dimensione della pagina del sistema. Per i sistemi x86 si tratta di pagine di 4KB.

Le zone possono essere liberate con *VirtualFree*.

Per utilizzare una zona riservata e' necessario che il programmatore ne assegni della memoria fisica, questo viene sempre effettuato con la funzione *VirtualAlloc*.

La memoria fisica

In windows 3.1 la memoria fisica era considerata la RAM della macchina. Successivamente venne aggiunto un supporto alla RAM sottoforma di file di swap e venne creato un algoritmo di memoria virtuale a pagine. Questo sistema si e' sviluppato fino ad oggi, con alcune migliorie.

Innanzitutto , quando si crea un processo, il sistema apre il file exe e usa lo stesso file come immagine della pagina di memoria in cui allocare il codice eseguibile. Questo vale per i file .EXE e i file .DLL; questo meccanismo permette di avere file di swao piu' piccoli e di condividere il codice e le DLL tra le applicazioni.

Questo meccanismo non viene eseguito se si carica il file da un floppy. Viene invece eseguito nel caso si carichi il file da un CD-ROM.

Ottenere informazioni sulla memoria

E' possibile ottenere informazioni sulla memoria mediante la funzione:

```
VOID GetSystemInfo( LPSYSTEM_INFO lpSystemInfo );
```

E' necessario passare a questa funzione una struttura SYSTEM_INFO che verra' riempita dal sistema:

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD dwOemId;
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD dwPageSize; //dimensione della pagina
    LPVOID lpMinimumApplicationAddress; //indirizzo minimo utilizzabile
    LPVOID lpMaximumApplicationAddress; //indirizzo massimo utilizzabile
    DWORD dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity; //granularita' del sistema
    WORD wProcessorLevel;
    WORD wProcessorRevision;
} SYSTEM_INFO;
```

E' possibile avere informazioni dinamiche sullo stato attuale della memoria:

```
VOID GlobalMemoryStatus( LPMEMORYSTATUS lpBuffer );
```

Anche in questo caso verra riempita una struttura MEMORYSTATUS, nella quale e' necessario riempire solamente il campo dwLength con sizeof(MEMORYSTATUS)

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength; // sizeof(MEMORYSTATUS)
    DWORD dwMemoryLoad; // memoria utilizzata dal gestore della memoria
    DWORD dwTotalPhys; // byte della RAM esistente (al netto di 598.016 byte)
    DWORD dwAvailPhys; // byte di RAM disponibile
    DWORD dwTotalPageFile; // byte massimi nel file di paginazione
    DWORD dwAvailPageFile; // byte disponibili nel file di paginazione
    DWORD dwTotalVirtual; // byte privati per ogni spazio di indirizzamento
    DWORD dwAvailVirtual; // byte privati disponibili nello spazio di indirizzamento
} MEMORYSTATUS, *LMEMORYSTATUS;
```

Esiste una funzione con la quale e' possibile conoscere lo stato di una qualunque casella di memoria nello spazio di indirizzamento privato di un processo:

```
DWORD VirtualQuery( LPCVOID lpAddress,  
                   PMEMORY_BASIC_INFORMATION lpBuffer,  
                   DWORD dwLength );
```

lpAddress e' l'indirizzo virtuale richiesto

dwLength e' la lunghezza della struttura

Il sistema riempie la struttura MEMORY_BASIC_INFORMATION

```
typedef struct _MEMORY_BASIC_INFORMATION {  
    PVOID BaseAddress;           // uguale a lpAddress  
    PVOID AllocationBase;       // indirizzo di base della zona  
    DWORD AllocationProtect;    // attributo di protezione  
    DWORD RegionSize;          // grandezza in byte della regione  
    DWORD State;                // stato della zona(libero, riservato)  
    DWORD Protect;              // attributo di protezione  
    DWORD Type;                 // tipo di memoria (fisica, mappata, immagine)  
} MEMORY_BASIC_INFORMATION;  
typedef MEMORY_BASIC_INFORMATION *PMEMORY_BASIC_INFORMATION;
```

Usare memoria virtuale nelle applicazioni

Win32 offre 3 meccanismi per manipolare grosse aree di memoria:

- 1) la memoria virtuale
- 2) i file mappati in memoria (utile per condividere dati tra processi)
- 3) gli heap

Analizziamo il primo metodo.

E' possibile allocare la memoria virtuale con:

```
LPVOID VirtualAlloc( LPVOID lpAddress,  
                    DWORD dwSize,  
                    DWORD flAllocationType,  
                    DWORD flProtect );
```

lpAddress contiene l'indirizzo da allocare. E' possibile passare NULL in modo che il sistema allochi automaticamente la memoria dove e' piu' opportuno (sempre nella zona privata del processo).

dwSize e' la dimensione richiesta in byte, verra' preso in considerazione un multiplo della pagina .

flAllocationType indica se si desidera riservare una zona o impegnare memoria fisica.

flProtect indica l'attributo di protezione (sono costanti predefinite).

La funzione restituisce il puntatore alla zona allocata (eventualmente arrotondato alla granularita' prevista).

Se non e' possibile allocare la zona la funzione restituisce NULL.

E' possibile seguire diverse modalita':

- riservare la memoria e impegnarla immediatamente (flAllocationType=MEM_RESERVE | MEM_COMMIT)
- riservare la memoria (flAllocationType=MEM_RESERVE) e impegnarla in un secondo momento (flAllocationType=MEM_COMMIT)
In questo caso e' possibile impegnare anche un sottoinsieme della memoria riservata utilizzando i due parametri lpAddress e dwSize.

In ogni caso e' necessario richiamare la funzione VirtualAlloc.

Mediante questo doppio meccanismo (denominato To reserve and To commit) e' possibile risolvere problemi di grosse matrici utilizzate in modo sparso (come nel caso di un foglio elettronico). In questo caso e' necessario organizzare l'applicazione in modo da impegnare e liberare ogni volta zone diverse della memoria riservata.

Per disimpegnare la memoria impegnata si usa:

```
BOOL VirtualFree( LPVOID lpAddress,  
                 DWORD dwSize,  
                 DWORD dwFreeType );
```

lpAddress contiene l'indirizzo da deallocare.

dwSize lunghezza area da disimpegnare (sempre = 0 per dereservare la memoria)

dwFreeType=MEM_DECOMMIT per disimpegnare la memoria

dwFreeType=MEM_DEREGISTER per deriservare la memoria

Anche in questo caso si lavora con la granularita' del sistema e la dimensione della pagina.

Decidere nel caso di una applicazione come il foglio elettronico quando disimpegnare la memoria non e' semplice.

FILES MAPPATI IN MEMORIA

I files mappati in memoria consentono di riservare una zona dello spazio di indirizzamento e di impegnare per questa zona della memoria fisica. La differenza rispetto alla VirtualAlloc consiste nel fatto che la memoria fisica proviene dal file stesso che si trova già su disco, anziché il file di paginazione del sistema.

I files mappati in memoria vengono impiegati per tre differenti utilizzi:

- Il sistema usa i file mappati in memoria per caricare ed eseguire EXE e DLL. Questo metodo permette di risparmiare spazio nel file di paginazione e tempo nella esecuzione.
Infatti al momento dell'esecuzione di CreateProcess (o di LoadLibrary per le DLL) il sistema riserva un'area di memoria sufficientemente grande per ospitare il codice e mappa questa memoria nello stesso file EXE o DLL del disco.
Servendosi dei files mappati in memoria più istanze in esecuzione della stessa applicazione possono condividere lo stesso codice (come abbiamo visto nella prima lezione).
- E' possibile servirsi di files mappati in memoria per accedere ad un file di dati su disco.
- E' possibile usare i files mappati in memoria per condividere i dati tra processi.

Esempio di utilizzo dei file mappati in memoria per invertire i byte di un file. Per risolvere questo problema ci sono 4 approcci:

1. Assegno un blocco di memoria grande quanto il file
Carico il file
Inverto i byte in memoria
Riscrivo tutto il file

Metodo semplice, non utilizzabile per un file di 2GB

2. Creo un file di destinazione
Alloco 8K di memoria
Carico gli ultimi 8K del file , li inverto e li scrivo nel file di destinazione itero fino ad arrivare all'inizio del file

Metodo più complesso, soprattutto nel caso di file non multipli di 8K
Può portare ad un uso eccessivo dello spazio su disco
Usa veramente poca memoria

3. Alloco 2 blocchi di 8K di memoria
Carico gli ultimi 8K del file nel primo blocco e i primi 8K nel secondo blocco, inverto i byte di entrambi i blocchi e li riscrivo nello stesso file originale itero fino ad arrivare nel centro del file

Metodo più complesso, soprattutto nel caso di file non multipli di 8K
Usa poca memoria

5. Con i memory mapped files si segue questa procedura:
Apro il file e si comunica al sistema di riservare una zona dello spazio di indirizzamento virtuale.
Si dice al sistema di mappare il primo byte del file con il primo byte della zona

riservata.

A questo punto si puo' usare il file come se fosse caricato in memoria, pensa il sistema ad effettuare le operazioni di caching sul file. Se il file finisse con un byte #0 si potrebbe usare `_strrev` del C per invertire i byte.

Usare memory mapped files nelle applicazioni

Per utilizzare un file mappato in memoria, e' necessario eseguire 3 passi:

1. Creare o aprire un oggetto kernel di tipo file

```
HANDLE CreateFile( LPCTSTR lpFileName,           //nome del file
                  DWORD dwDesiredAccess,        //tipo di accesso (read,write)
                  DWORD dwShareMode,           //come condividere il file
                  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                  DWORD dwCreationDistribution,
                  DWORD dwFlagsAndAttributes,
                  HANDLE hTemplateFile );
```

N.B: in Win32 si usa `CreateFile` anche per aprire i files (la `OpenFile` e' solo mantenuta per compatibilita' con Win16)

ATTENZIONE : `CreateFile` e' l'unica funzione Win32 che in caso di mancata creazione restituisce l'handle del file `0xFFFFFFFF (INVALID_HANDLE_VALUE)` e non `NULL`.

2. Creare un oggetto kernel di tipo mappa-file

```
HANDLE CreateFileMapping( HANDLE hFile,           //handle restituito da CreateFile
                          LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
                          DWORD flProtect,        //attributi di protezione della memoria
                          DWORD dwMaximumSizeHigh, //dimensioni max file(64 bit)
                          DWORD dwMaximumSizeLow,
                          LPCTSTR lpName );       //nome dell'oggetto (condivisibile)
```

3. Dire al sistema di mappare tutto o parte del file nell'oggetto mappa-file. In questo modo si assegna all'oggetto mappa-file una vista (view) del file vero e proprio.

```
LPVOID MapViewOfFile( HANDLE hFileMappingObject, //handle da CreateFileMapping
                      DWORD dwDesiredAccess, //di nuovo il tipo di accesso
                      DWORD dwFileOffsetHigh, //primo byte da mappare (64 bit)
                      DWORD dwFileOffsetLow,
                      DWORD dwNumberOfBytesToMap ); //lunghezza da mappare
```

Per liberare il file mappato in memoria e' necessario:

1. comunicare al sistema di demappare l'oggetto mappa-file

```
BOOL UnmapViewOfFile( LPCVOID lpBaseAddress );
```

`lpBaseAddress` deve essere lo stesso restituito da `MapViewOfFile`.

N.B: e' importante richiamare questa funzione altrimenti la zona di memoria non sara' liberata fino alla terminazione del processo.

2. chiudere l'oggetto mappa-file e chiudere l'oggetto file.

CloseHandle

N.B: e' buona abitudine pulire sempre l'handle con CloseHandle.

Usare memory mapped files per condividere dati tra processi

Per fare questo e' sufficiente che due processi creino due viste diverse dello stesso file, automaticamente condivideranno le stesse pagine di memoria virtuale.

Un processo chiamera' CreateFileMapping passando il nome "miofilemapp", dopodiche' se non esiste gia' un oggetto mappa-file con lo stesso nome lo crea, altrimenti restituisce l'handle dell'oggetto gia' esistente.

E' possibile in alternativa usare OpenFileMapping (come per gli altri oggetti kernel).

Un altro processo utilizzando lo stesso nome potra' facilmente condividere il file mappato.

Questo sistema non e' molto comodo quando si devono trasferire dati creati da un processo e trasferirli ad un altro processo. Sarebbe necessario tutte le volte creare un file su disco. La microsoft per ovviare a questo inconveniente ha permesso la creazione di memory mapped file mappati nel file di paginazione.

Questo metodo e' identico al precedente, solamente non si richiama CreateFile, e si chiama CreateFileMapping passando come handle del file 0xFFFFFFFF.

In questo modo si comunica al sistema di voler utilizzare un memory mapped file nel file di paginazione.

Due processi possono condividere questo oggetto come se fosse una pura e semplice zona di memoria virtuale.

ATTENZIONE : non dare in ingresso immediatamente l'handle di un file ottenuto da CreateFile senza controllare se e' uguale a 0xFFFFFFFF altrimenti non si riuscirà a capire se CreateFile ha fallito.

Quando l'oggetto mappa-file viene distrutto tutti i dati vengono persi !!!!

-> Un metodo alternativo per effettuare la stessa operazione e' stato creato con il messaggio WM_COPYDATA del quale vi ho fornito un esempio.

N.B: Solamente nel caso in cui si condivida un memory mapped file e' possibile non impegnare tutta la memoria riservata, ma utilizzare la metodologia To reserve and To commit vista precedentemente (vedere pag 278-279 Testo Ritchter).

Aprile 1999

LIBRERIE DI CONCATENAMENTO DINAMICO IN WINDOWS 95 - NT

*Ing. Stefano Riccio
stefaric@tiscalinet.it*

LE DLL

Le DLL sono la pietra angolare dei sistemi Windows fin dalla prima versione.

Le DLL sono costituite da un gruppo di funzioni autonome che qualsiasi applicazione puo' utilizzare.

Per creare DLL e' sufficiente indicare al linker il commutatore /DLL, questo viene fatto dal Wizard Win32 DLL di Visual C++.

Perche' una applicazione (o una DLL) chiami le funzioni contenute in una DLL, l'immagine del file della DLL deve prima essere mappata nello spazio di indirizzamento del processo chiamante. Per fare questo ci sono due metodi:

1. Concatenamento implicito in fase di caricamento

Quando si fa il link della applicazione, al linker sono specificati l'elenco delle funzioni della DLL. In questo caso quando il link incontra l'utilizzo di una funzione ne inserisce nel file EXE l'immagine. Cosi' il loader del sistema operativo e' in grado di caricare tutte le DLL nello spazio di indirizzamento.

Le DLL vengono cercate in questo ordine:

- la directory che contiene il file EXE
- la directory corrente del processo
- la directory di sistema di Windows
- la directory di Windows
- le directory della variabile PATH

Se non trova la DLL emette un messaggio di errore.

2. Concatenamento esplicito in fase di esecuzione

In questo caso l'immagine del file DLL puo' essere mappato quando uno dei thread della applicazione chiama la funzione *LoadLibrary* o *LoadLibraryEx*

```
HINSTANCE LoadLibrary( LPCTSTR lpLibFileName );
```

```
HINSTANCE LoadLibraryEx( LPCTSTR lpLibFileName,  
                          HANDLE hFile,  
                          DWORD dwFlags );
```

La funzione *LoadLibraryEx* e' piu' complessa e permette di modificare, mediante il parametro *dwFlags*, il comportamento standard. Ad esempio permette di non eseguire la funzione di entrata nella DLL (utile per le DLL delle risorse), modifica la ricerca nelle directory ecc...

Il valore *HINSTANCE* rappresenta l'istanza della DLL.

Quando si carica una DLL con *LoadLibrary* si puo liberare la memoria con

```
BOOL FreeLibrary( HMODULE hLibModule );
```

passando l'istanza ottenuta precedentemente.

Nelle DLL e' possibile utilizzare variabili globali, ma queste non possono essere condivise tra piu' istanze della stessa DLL (esattamente come i file EXE).

Quest'ultima cosa era permessa invece in Windows 16 bit in quanto ogni DLL aveva il proprio segmento dati e quindi accessibile a tutti i processi che caricavano quella DLL. In Win32 una DLL non dispone piu' di memoria propria, ma utilizza sempre e solo lo spazio di indirizzamento del processo chiamante.

Altre funzioni utili sono le seguenti:

```
HMODULE GetModuleHandle( LPCTSTR lpModuleName );
```

per sapere se una DLL e' caricata nello spazio di indirizzamento.

```
DWORD GetModuleFileName( HMODULE hModule,  
                          LPTSTR lpFilename,  
                          DWORD nSize );
```

per sapere il nome della DLL conoscendo l'handle.

La funzione ingresso/uscita di una DLL

A differenza delle DLL Win16 le DLL Win32 hanno una sola funzione di entrata/uscita, denominata DllMain.

In win 16 c'erano due funzioni: LibMain e WEP, inoltre era necessario portarsi dietro un pezzo di codice in Assembler. Tutto questo e' stato eliminato poiche' API Win32 deve essere un codice portabile su altri processori e non si possono inserire codici macchina nei sorgenti.

```
BOOL WINAPI DllMain (HINSTANCE hinstDll, DWORD fdwReason, LPVOID lpvReserved) {  
    switch (fdwReason) {  
  
        case DLL_PROCESS_ATTACH:  
            // DLL is attaching to the address  
            // space of the current process.  
  
            break;  
  
        case DLL_THREAD_ATTACH:  
            // A new thread is being created in the current process.  
  
            break;  
  
        case DLL_THREAD_DETACH:  
            // A thread is exiting cleanly.  
  
            break;  
  
        case DLL_PROCESS_DETACH:  
            // The calling process is detaching  
            // the DLL from its address space.  
  
            break;  
    }  
  
    return(TRUE);  
}
```

Questa funzione viene richiamata dal sistema, tramite il thread principale (concatenamento implicito) o un altro thread (concatenamento esplicito) passando l'istanza della DLL (la quale e' buona norma conservarsi in una variabile globale) e un identificativo del tipo di operazione che si sta eseguendo (fdwReason).

Vediamo le 4 possibilita':

1. DLL_PROCESS_ATTACH

Quando una DLL viene mappata per la prima volta in uno spazio di indirizzamento di un processo, il sistema chiama DllMain passandole DLL_PROCESS_ATTACH. Cio' si verifica una volta per ogni processo in cui si carica quella DLL.

In risposta a questa chiamata il codice dovrebbe inizializzare le strutture utili al processo. E' importante restituire TRUE se tutte le inizializzazioni sono riuscite, FALSE altrimenti (in modo che venga rilasciata la DLL).

2. DLL_PROCESS_DETACH

Quando una DLL viene tolta dallo spazio di indirizzamento viene chiamata DllMain con questo parametro. In questo caso il codice deve distruggere gli oggetti creati e la memoria allocata.

ATTENZIONE : Se si restituisce FALSE alla chiamata DLL_PROCESS_ATTACH viene comunque richiamata DllMain con fdwReason= DLL_PROCESS_DETACH, non bisogna quindi liberare zone di memoria non allocate.

3. DLL_THREAD_ATTACH

Ogni volta che viene creato un thread (CreateThread) il sistema esamina tutte le DLL mappate nello spazio di indirizzamento e per ognuna richiama DllMain passando questo parametro.

Non viene eseguita questa chiamata per thread gia' attivi al momento del caricamento di una DLL.

Questa chiamata non viene fatta per il thread principale, per il quale esiste quella del processo.

4. DLL_THREAD_DETACH

Quando termina un thread viene invocata la funzione DllMain di ogni DLL caricata nello spazio di indirizzamento con questo parametro (Tranne nel caso di chiamata TerminateThread).

N.B: questo sconsiglia in linea generale di usare TerminateThread, e comunque usarla con molta attenzione.

Questa chiamata non viene fatta per i thread attivi al momento dello scaricamento di una DLL e ovviamente per il thread principale.

ATTENZIONE : Le chiamate alla DllMain vengono serializzate dal sistema mettendo in sonno i thread che attendono. Quindi bisogna stare attenti a non creare situazioni di deadlock.

E' possibile utilizzare la funzione DisableThreadLibraryCalls che annulla le due chiamate relative al processo.

Le funzioni di una DLL

Le funzioni di una DLL che si vogliono rendere disponibile ai processi ospitanti si devono **ESPORTARE**. Con Win32 e' possibile esportare funzioni e variabili globali.

Per esportare le funzioni e' sufficiente premettere alla dichiarazione di funzione o variabile la parola chiave:

```
_declspec(dllexport)
```

Quando il compilatore incontra questa parola inserisce delle informazioni aggiuntive al file OBJ. Successivamente il linker inserisce nella DLL un elenco di simboli esportabili con associato l'indirizzo di partenza della funzione o della variabile.

Se si vuole utilizzare, da una applicazione le funzioni di una DLL, e' necessario dichiarare le funzioni indicando che provengono da una DLL mediante la seguente parola chiave:

```
_declspec(dllimport)
```

Questo informa il compilatore di andare a cercare la funzione tra gli elenchi di simboli esportati nelle DLL. Anche nel file EXE dell'applicativo e' contenuta una tabella di importazione.

E' possibile ottenere dinamicamente l'indirizzo di una funzione in una DLL utilizzando

```
FARPROC GetProcAddress( HMODULE hModule, //handle ottenuto da LoadLibrary  
                        LPCSTR lpProcName ); //nome della funzione
```

Questa funzione viene usata spesso insieme a LoadLibrary nel caso di concatenamento esplicito.